Titre: Title:	Réalisation d'une plateforme de développement rapide de systèmes sur puce basée sur UML
Auteur: Author:	Alexandre Chureau
Date:	2005
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Chureau, A. (2005). Réalisation d'une plateforme de développement rapide de systèmes sur puce basée sur UML [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7603/

UNIVERSITÉ DE MONTRÉAL

RÉALISATION D'UNE PLATEFORME DE DÉVELOPPEMENT RAPIDE DE SYSTÈMES SUR PUCE BASÉE SUR UML

ALEXANDRE CHUREAU
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
JUIN 2005

Your file Votre référence
ISBN: 978-0-494-16768-7
Our file Notre référence
ISBN: 978-0-494-16768-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

RÉALISATION D'UNE PLATEFORME DE DÉVELOPPEMENT RAPIDE DE SYSTÈMES SUR PUCE BASÉE SUR UML

présenté par: CHUREAU Alexandre

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

Mme NICOLESCU Gabriela, Ph.D., présidente

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. ABOULHAMID El Mostapha, Ph.D., membre et codirecteur de recherche

M. BOIS Guy, Ph.D., membre

À Robert Dubois, France Rochette et Yvon Savaria, humanistes

REMERCIEMENTS

J'aimerais remercier les personnes suivantes :

M. Yvon Savaria et M. El Mostapha Aboulhamid pour leur confiance et leurs conseils ;

Mme Gabriela Nicolescu et M. Guy Bois pour leur évaluation de ce mémoire ;

M. François Gagnon, M. Claude Thibeault ainsi que toute l'équipe PROMPT-MAME de l'École de technologie supérieure qui ont fourni un environnement d'expérimentation exceptionnel ;

Le personnel du GRM pour son support technique et logistique ;

Ainsi que les organismes suivants :

Logiciels et formation : Société canadienne de Microélectronique, IBM-Rational ;

Soutien financier : Micronet R&D, Regroupement Stratégique en Microélectronique du Québec (ReSMiQ), PROMPT-Québec.

RÉSUMÉ

L'objectif de cette recherche est d'utiliser la spécification d'interfaces et d'autres techniques du génie logiciel afin de maîtriser la complexité du design des systèmes sur puce. Un modèle UML est utilisé comme point de départ du design afin de capturer les spécifications d'interfaces des principaux composants du système à un haut niveau d'abstraction. Les composants sont représentés sous forme de capsules UML détaillées qui établissent des contrats d'interface de façon indépendante de l'implantation. L'utilisation de communications par messages entre les capsules permet une interconnexion lâche des composants qui facilite l'exploration architecturale. Le raffinement itératif des capsules est ensuite effectué afin d'obtenir d'abord un modèle logiciel exécutable du système pour ensuite procéder à l'implantation du modèle en exploitant les ressources matérielles cibles. Parallèlement, un environnement de vérification est développé pour fournir des stimuli au modèle exécutable afin d'observer son comportement. Le couple modèle exécutable - environnement de vérification forme un prototype fonctionnel du système qui demeurera cohérent tout au long du raffinement. Chaque niveau d'abstraction du prototype constitue une plateforme où différents composants réels ou virtuels provenant d'une bibliothèque peuvent être agencés et paramétrés en fonction d'une application donnée. Afin d'assurer la cohérence du prototype, malgré la présence de composants réels et virtuels, logiciels et matériels, une capsule de transaction est développée. Présentant un aspect externe régulier, cette capsule implante un mécanisme de communication basé sur le protocole TCP/IP qui permet l'intégration de composants hétérogènes à l'intérieur du prototype. Ces techniques sont mises en oeuvre pour la spécification et l'implantation d'un égaliseur adaptatif linéaire pour une application de radio réalisée par logiciel. La structure de l'égaliseur est modélisée à l'aide de capsules UML et son comportement est modélisé à l'aide de machines à états détaillées en langage C++. Un environnement de vérification est développé à l'aide de l'outil Simulink

afin de fournir des données de radio numérique et d'observer les résultats produits par l'égaliseur. Un mécanisme de transaction est inséré dans chacun des modèles afin d'établir une communication transparente entre les composants simulés et les composants exécutés. Le prototype fonctionnel ainsi formé est validé puis raffiné et profilé afin de démontrer l'utilité des techniques dans la maîtrise de la complexité du design. Le langage UML s'avère adéquat pour l'organisation structurelle des composants d'un système sur puce de même que pour une modélisation à haut niveau d'abstraction des communications, ce qui permet une validation de la spécification tôt dans le design. La limite pratique d'un modèle UML est observée lorsqu'il s'agit de paralléliser l'exécution d'un modèle hétérogène, ce qui exige des mécanismes de communication de bas niveau performants et une précision temporelle difficilement atteignable au niveau du modèle.

ABSTRACT

The objective of this research is to handle system-on-chip design complexity by using interface-based design and other software engineering techniques. A UML model is used as a starting point to capture the specifications of the main components interfaces at a high level of abstraction. These components are represented as capsules that are implementation-independent and detailed with performance data to establish the component's interface contract. Using message-based communications allows loose coupling of the components and eases design-space exploration. The capsules are iteratively refined into an executable software model and then mapped to target hardware resources. A verification environment is developed in parallel in order to provide stimuli to the executable model and observe its behavior. The executable model together with the verification environment form a functional prototype of the system that will serve as a vehicle to the refinement process. Each abstraction level in the prototype represents a platform where components, available from a library, are interconnected and configured to answer the needs of a specific application. In order to preserve coherence while allowing heterogeneous components to be integrated in the prototype, a transactor capsule is developed. This capsule presents a standard interface but implements a communication mechanism based on TCP/IP that resolves interactions between different models of computation. These techniques are put in practice for the specification and implementation of a linear adaptive equalizer for a software-defined radio application. The structure of the equalizer is specified using capsules and the behavior is modeled with state machines detailed with C++ code. A verification environment is developed with Simulink to provide realistic digital radio data and observe the equalizer output. A transaction mechanism is introduced in each model in order to allow seamless communications between executed and simulated components. The functional prototype that is obtained is validated, then refined and profiled in

order to demonstrate the utility of the techniques in handling design complexity. The UML language proves to be efficient for structural organization of functional components and high-level modeling of communications, which allow the specification to be validated early in the design. The practical limit of a UML model is observed when parallelizing processing units of a heterogeneous design. This requires efficient low-level communication mechanisms for synchronization and a timing precision that is hardly possible at the model level.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES

AGC :	Automatic Gain Control
ASIP :	Application-Specific Instruction-Set Processor
ASSP :	Application-Specific Standard Product
DSP :	Digital Signal Processing
FFT :	Fast Fourier Transform
FIFO :	First In First Out
FIR :	Finite-Impulse Response
FPGA :	Field-Programmable Gate Array
GDSII :	Graphic Design System II
HDL :	Hardware Description Language
ISS :	Instruction Set Simulator
HVL :	Hardware Verification Language
MDA :	Model-Driven Architecture
MPSoC :	Multiprocessor System-on-Chip
NoC :	Network-on-Chip
OCL :	Object Constraint Language
PIM :	Platform-Independent Model
PSM :	Platform-Specific Model
RRL :	Radio réalisée par logiciel
RTL :	Register-Transfer Level
SoC :	System-on-Chip
SoPC :	System-on-a-Programmable-Chip
UML :	Unified Modeling Language
VHDL :	Very High Speed HDL

LISTE DES ANNEXES

INTRODUCTION

Chaque circuit intégré répond de façon unique aux besoins d'une application. Cette conception sur mesure assure une performance optimale du circuit mais requiert un effort de développement considérable. En effet, les techniques de fabrication microélectronique actuelles permettent d'intégrer un système ordiné complet sur une seule puce, ou système sur puce, ouvrant la porte à l'intégration d'applications très élaborées. Ces circuits contiennent habituellement des éléments de traitement, des éléments de mémoire de même qu'un système de communication sur puce.

Au contraire de l'industrie des ordinateurs personnels, qui repose sur des plates-formes de conception matérielle bien établies, l'industrie de la microélectronique doit réinventer chaque fois en tout ou en partie l'infrastructure d'un nouveau système. L'inflexibilité des circuits spécifiques est la principale cause de ce lourd travail, qui menace la productivité des méthodes de conception depuis quelques années. Les contraintes matérielles de même que les requis de performance imposés par l'application justifient le développement de circuits dédiés à un certain type de traitement, peu flexibles mais très performants. Si la conception de ces circuits ne se fait pas dans un cadre dicté par leur futur environnement opérationnel, leur intégration dans le système final représente un certain degré de risque. Dans le cas d'un système sur puce, ce sont des dizaines, parfois des centaines de circuits hétérogènes qui sont assemblés afin de réaliser une application, multipliant d'autant le degré de risque associé au développement.

De par leur nature hétérogène et compacte, les systèmes sur puce sont un véritable casse-tête de vérification et de validation. Ces activités consomment à elles seules la majorité du temps de développement d'un nouveau circuit, près de 80% selon plusieurs sources (Drechsler, 2003; Anderson, 2004). Des tests doivent être

effectués en continu depuis la spécification du circuit jusqu'à son empaquetage afin de détecter les erreurs le plus tôt possible dans le design. À cette fin, l'utilisation d'un prototype exécutable en C++ ou en VHDL facilite le traitement et l'analyse des vecteurs de test. Mais la cohérence entre un prototype et son implantation est ultimement mise en jeu par les enjeux physiques des technologies de fabrication. Depuis le passage sous la barre des 0.25 microns, la réduction de l'échelle n'affecte plus tous les paramètres de fabrication de la même façon. Les courants de fuite, par exemple, freinent la réduction des tensions de seuil. Le gain en performance a donc commencé à décroître relativement au facteur de réduction de l'échelle, alors que d'autres difficultés apparaissaient : la résistivité croissante des interconnexions, la dissipation de chaleur, la sensibilité au bruit, l'électro-migration, etc. (Cohn, 2003).

En somme, les défis d'interconnexion, de vérification et la pression croissante pour un temps de mise en marché plus court des nouveaux circuits poussent les méthodes de conception à passer à des niveaux d'efficacité supérieurs. Suivant l'évolution naturelle vers des niveaux d'abstraction plus élevés, la conception fait maintenant appel à une réutilisation de composants pré-conçus de plus grande envergure et programmables. La granularité du design est donc passée du niveau portes et mémoires au niveau de modules IP (*Intellectual Property*) réutilisables. Cependant, les défis d'interconnexion ne s'en sont retrouvés que déplacés au niveau de l'interface entre ces modules. En réponse au problème récurrent des interconnexions, l'industrie a adopté la conception basée plate-forme, où le design est démarré à partir d'assemblages flexibles de modules visant un éventail prédéfini d'applications.

L'orientation des méthodes de conception matérielle vers les méthodes de conception logicielle est évidente. Il est possible d'associer le concept des plates-formes matérielles aux nombreuses bibliothèques de composants spécialisés disponibles pour le développement logiciel, abstraction faite des contraintes matérielles et de performance. En matériel, ces contraintes se traduisent par la présence d'éléments

hétérogènes dans une plate-forme, décrits à des niveaux d'abstraction différents ou voués à des traitements de nature différente. En proposant une certaine façon d'interconnecter ces composants, une plate-forme ne résout que partiellement le problème des interconnexions. Il y a un véritable besoin de s'élever plus haut dans les niveaux d'abstraction, et ceci doit inclure une représentation efficace des interfaces entre les composants hétérogènes.

L'objectif de cette recherche est de décrire une nouvelle approche à la conception de systèmes électroniques qui facilite le développement et l'intégration de composants multiples et hétérogènes et ce indépendamment de leur niveau d'abstraction ou de la nature de leur traitement. Une technique empruntée au génie logiciel est suggérée afin d'y parvenir : l'encapsulation des détails d'implantation à l'intérieur d'interfaces spécifiées à l'aide d'un langage indépendant.

L'hypothèse est faite que la caractérisation des interfaces d'un système à l'aide d'un langage indépendant permet de maîtriser la complexité du design en découplant fonctionnalité et architecture. À cette fin, le langage UML constitue un choix privilégié pour son pouvoir expressif, son indépendance face à toute implantation et sa flexibilité. Les différents niveaux d'abstraction impliqués dans un flot de design peuvent être encapsulés et projetés à un niveau "messages" commun à travers le système. Les concepts de *protocole* et de *capsules*, introduits récemment dans la spécification du langage UML, seront utilisés pour élaborer ce niveau.

Cette technique de caractérisation des interfaces constitue un élément méthodologique intéressant pour faciliter la conception basée plate-forme. Un design peut donc débuter avec l'ensemble des spécifications d'interface développées pour une famille d'applications cible. De nouvelles interfaces peuvent s'ajouter au besoin afin d'agrandir la bibliothèque des composants disponibles. La transformation et le raffinement des interfaces en infrastructure de communication sur puce complète le

cycle de développement d'un nouveau circuit à partir de la plate-forme.

Une méthode progressive est utilisée afin de valider l'hypothèse de recherche. Un premier composant simple est caractérisé sous forme d'un contrat d'interface. Différentes implantations du composant sont réalisées sous l'égide du même contrat afin de démontrer l'orthogonalisation des aspects fonction et architecture. Le concept d'interface est ensuite étendu afin de prendre en compte l'environnement d'insertion d'un composant et l'hétérogénéité qui peut en résulter. Cette nouvelle façon de spécifier les interfaces est appliquée à un niveau de complexité supérieur où deux composantes hétérogènes doivent communiquer et former un système cohérent. Le développement d'un système réel à ce stade permettra la comparaison des bénéfices et inconvénients de cette méthodologie par rapport à une méthode de conception conventionnelle. Une application de radio réalisée par logiciel sera implantée et mettra à contribution l'expertise du groupe de recherche MAME de l'École de technologie supérieure [1] dans ce domaine.

Le premier chapitre de ce mémoire fait un exposé des connaissances actuelles dans le domaine de la conception de systèmes sur puce, tant au niveau des technologies que des méthodologies de conception et de vérification. Le chapitre suivant propose une nouvelle méthodologie de conception qui met l'emphase sur la conception des interfaces d'un système sur puce par le biais de spécifications en UML. Le troisième chapitre fait état de l'application de la méthodologie à la conception d'un égaliseur destiné à une application de radio réalisée par logiciel. Le quatrième chapitre présente une analyse des bénéfices et inconvénients de la méthode de conception utilisée. Une discussion sur les améliorations possibles et les perspectives de recherche qui découlent de ce travail conclut le mémoire.

[1] Méthodologies et Architectures pour la Multiégalisation (MAME, 2005)

CHAPITRE 1

REVUE DE LITTÉRATURE

Le premier circuit intégré, fabriqué en 1958, comptait moins de dix transistors. Depuis, l'évolution des techniques de fabrication, permise d'une part par la réduction de l'échelle lithographique et d'autre part par un contrôle exact du processus et des matériaux, a fait exploser ce nombre. Outre l'augmentation du nombre de transistors, la réduction de l'échelle a également mené à une diminution de la consommation d'énergie par transistor ainsi qu'à l'accélération des vitesses d'horloge. Une référence actuelle en la matière, le processeur Pentium 4 d'Intel, basé sur une géométrie de 130 nm et disponible depuis l'an 2000, contient plus de 42 millions de transistors et fonctionne à une fréquence de 3.4 GHz (Intel Corporation, 2005a). Les techniques de fabrication les plus avancées travaillent actuellement avec des géométries de 90 nm, 65 nm et parfois 45 nm. Ces dimensions sont généralement réservées aux circuits numériques opérant à basse tension, alors que la géométrie de 130 nm continue d'être utilisée pour les circuits analogiques et mixtes (Lammers, 2005). L'édition 2004 du rapport de l'International Technology Roadmap for Semiconductors (ITRS)(Semiconductor Industry Association, 2004) confirme que cette diminution de l'échelle s'étendra encore sur plusieurs années (le rapport fait une estimation aussi optimiste que 7 nm en 2018 pour les microprocesseurs, en fonction des limites physiques connues de la technologie CMOS).

1.1 Évolution des techniques et méthodologies de conception

L'évolution des techniques d'intégration a entraîné avec elle une révolution des techniques de conception de systèmes numériques. Afin d'augmenter la productivité et la fiabilité du design, il fut nécessaire d'orchestrer le nombre croissant de fonctions intégrables par le biais de techniques de modélisation et de conception systématique. À la base, la modélisation permet un meilleur contrôle de la complexité du design par le choix d'un niveau de détail approprié. (Ashenden, 2002) fait état d'autres avantages qui découlent de cette élévation dans les niveaux d'abstraction. Il cite la possibilité de :

- Communiquer la compréhension des fonctions à l'usager du système ;
- Faciliter la vérification par le biais de simulations ;
- Permettre la vérification formelle d'un design ;
- Permettre la synthèse automatique ;
- Faciliter la réutilisation du design.

Dans le contexte d'un système électronique, trois domaines de modélisation distincts ont été identifiés : fonctionnel, structurel et géométrique (Gajski et Kuhn, 1983). Le domaine fonctionnel décrit le comportement d'un système en relation avec son environnement. Par domaine structurel, on entend l'ensemble des composants qui, interconnectés, forment un système. Enfin, le domaine géométrique contient les spécifications qui mèneront à une implantation physique du système. Chaque domaine de modélisation possède ses propres méthodes de transformation d'un niveau d'abstraction à l'autre. Par exemple, dans le domaine structurel, on fera abstraction des transistors en les regroupant en portes, qui seront elles-mêmes abstraites par un modèle à transfert de registres (Register-Transfer-Level).

Des langages de modélisation adaptés aux besoins spécifiques des circuits intégrés ont été développés pour la construction de modèles dans les domaines fonctionnel, structurel et géométrique. Les langages Verilog et VHDL sont certainement les plus connus de ces langages de modélisation. Ces langages offrent la possibilité de construire des modèles qui couvrent les domaines structurel et fonctionnel et ce, à un niveau d'abstraction variable. Ces langages permettent également la simulation et la transformation systématique d'un modèle d'un niveau d'abstraction à l'autre, comme l'illustre la Figure 1.1.

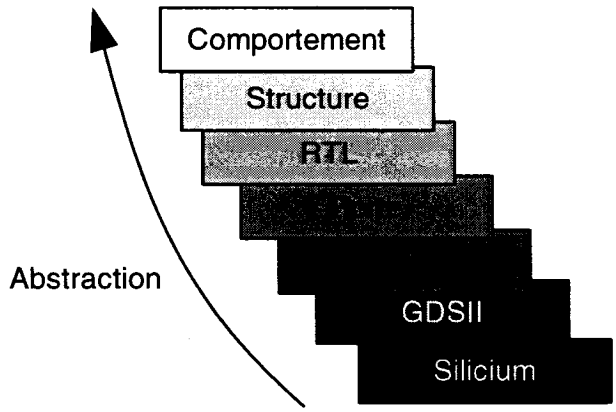

FIG. 1.1: Transformations itératives d'un modèle VHDL

Les outils de simulation et de synthèse utilisant le VHDL et le Verilog ont permis à la productivité du design de suivre de près l'évolution de la capacité d'intégration pendant un certain temps. Mais la poussée constante de la loi de Moore[1] ainsi que la pression du temps d'accès au marché ont mis à rude épreuve la productivité de ces outils. Ainsi, vers la fin des années 90, les organismes Semiconductor Industry Association et SEMATECH font état du problème de l'écart croissant en regard de la productivité des méthodes de conception (Semiconductor Industry Association, 1999). Cet écart de productivité met en évidence la nécessité d'automatiser

[1]Cette loi est en fait une prédiction faite en 1965 par Gordon Moore (et révisée subséquemment), à savoir que la densité et la performance des circuits intégrés est appelée à doubler tous les 18 à 24 mois (Intel Corporation, 2005b).

le processus de conception à partir de niveaux d'abstraction plus élevés, ainsi que le besoin de méthodes qui vont au-delà des langages.

Répondant au besoin d'automatisation, la synthèse à haut niveau, ou synthèse comportementale, part d'une spécification algorithmique d'un comportement et produit une représentation synthétisable au niveau transfert de registres qui implante ce comportement (McFarland et al., 1990). Il s'agit en fait d'établir un lien systématique entre les trois domaines de modélisation mentionnés par (Gajski et Kuhn, 1983), dans le but de raccourcir le cycle de design, diminuer le nombre d'erreurs, mieux explorer l'espace des designs possibles, etc. À partir d'une spécification de l'algorithme, le processus de synthèse produit une description du chemin de données sous la forme d'un réseau de registres, unités fonctionnelles, multiplexeurs et bus. Une description du chemin de contrôle peut également être produite lorsque nécessaire, sous la forme d'une machine à états finis. Les outils commerciaux et académiques de synthèse à haut niveau transforment actuellement des modèles développés en langages de 3e et 4e générations, comme les langages C/C++ (Mentor Graphics Corporation, 2005; Gupta et al., 2004), HandelC (Celoxica, 2005) ou Matlab (AccelChip, 2005), en représentation synthétisable. Leur véritable avantage réside dans la rapidité avec laquelle ils réalisent l'implantation d'un comportement, car il est généralement reconnu que les outils de synthèse à haut niveau ne produisent pas de meilleurs circuits qu'un concepteur expérimenté.

La synthèse de haut niveau est fortement liée à la notion de modèle de calcul. Un modèle de calcul est un patron d'exécution qui permet d'exprimer certains comportements de façon mathématique (Winskel et Nielsen, 1995). Ainsi, l'utilisation d'un modèle de calcul permet la définition de règles formelles pour la vérification du comportement et l'application de mécanismes de synthèse à haut niveau (Harel et Naamad, 1996). Parmi les principaux modèles de calculs qui s'appliquent à la conception de systèmes sur puce, on trouve la modélisation par événements discrets

(utilisé par le VHDL), la modélisation du temps continu (utilisé par le simulateur SPICE et par Simulink), les machines à états finis, les flots de données synchrones et les réseaux de Petri et de Kahn. Le choix d'un modèle de calcul dépend du type de traitement qui fait l'objet d'une modélisation. Certains modèles de calcul sont mieux adaptés à la modélisation du contrôle (ex. : machines à états finis) alors que d'autres sont plus appropriés pour la modélisation d'un flot de données défini et régulier (ex. : modèle des flots de données synchrones). Différents modèles de calcul peuvent être combinés afin de représenter les composants hétérogènes d'un système ou encore spécialisés afin de modéliser certains aspects d'un traitement. Ainsi, les diagrammes d'états de Harel, connus sous le nom de *statecharts*, sont une extension de la machine à états finis qui permet de modéliser la hiérarchie et la concurrence d'un traitement (Harel, 1987). (Jantsch, 2005) fait une distinction entre les différents modèles de calcul sur la base de leur traitement du temps. Ainsi, quatre niveaux de précision sont définis : le temps continu, le temps discret, le temps cadencé (clocked time) et la causalité. La conciliation des différents niveaux de précision du temps à l'intérieur d'un modèle hétérogène unique représente un défi important.

Du côté méthodologique, les progrès tendent vers une organisation plus efficace et une formalisation des processus de design et de vérification, comme le témoigne (Keating et Bricaud, 2002) et (Bergeron, 2003). Par exemple, les méthodes basées composants, telle que celles présentées dans (Cohen, 2000) et (Cesario et al., 2002), tentent de capturer les spécifications d'un système en terme d'éléments de fonctionnalité. Des systèmes complexes sont assemblés à partir de composants plus simples aux fonctions et caractéristiques bien définies, encourageant la réutilisation et facilitant la vérification. Cette approche de type ascendante peut être vue comme complémentaire à l'approche descendante des techniques présentées plus haut.

La réutilisation d'éléments préconçus s'est révélée être un élément méthodologique essentiel à la productivité du design qu'il faut mettre en pratique tout au long du processus de design, de vérification et de test (Jacome et Peixoto, 2001). Mise en oeuvre sous forme de bibliothèques de cellules normalisées (standard cell libraries) dès les débuts de l'automatisation du design, la réutilisation a successivement été projetée à des niveaux d'abstraction supérieurs : *compiled cells*, *macrocells*, *megacells* et *Intellectual Property*. (Chang et al., 1999) fait état de l'évolution des portfolio de réutilisation, du niveau personnel et opportuniste au niveau composant virtuel et planifié. Ce dernier niveau marque le gain de productivité le plus élevé en traçant clairement une ligne entre la création et l'intégration. Ainsi, plusieurs environnement de conception reposent sur une bibliothèque de composants pré-caractérisés, pré-vérifiés et pouvant être décrits à différents niveaux d'abstraction. Ces composants forment des noyaux conçus pour travailler ensemble et dont l'indépendance face à l'implantation varie. La société Xilinx, par exemple, a mis sur pied la bibliothèque LogiCore et le système CORE Generator, où l'on retrouve une panoplie de noyaux optimisés pour les FPGA Xilinx et qui réalisent une variété de tâches (FFT, FIFO, Cordic, encodeur et décodeur CDMA2000, etc.). La commercialisation de ces composants sous forme de modules de propriété intellectuelle et leur regroupement en bibliothèques accessibles en ligne (voir www.opencores.org et www.design-reuse.com) illustrent bien la maturité et l'importance qu'a atteint l'aspect réutilisation du design. Le consortium SPIRIT (www.spiritconsortium.com), qui regroupe des développeurs d'outils de conception, concepteurs de modules IP et fabricants de composants semi-conducteurs, a mis sur pied la spécification SPIRIT 1.0 pour faciliter l'encapsulation, l'échange et la réutilisation du design (Design and Reuse, 2004). La spécification repose sur le XML (Extensible Markup Language) afin de décrire les propriétés d'un module IP et de rendre les outils de conception, de vérification, de simulation et de synthèse compatibles entre eux.

La diminution de l'échelle et le progrès des techniques de fabrication ont toutefois poursuivi leur cours, ouvrant la porte à l'intégration de systèmes hétérogènes complexes, les systèmes sur puce. Ces circuits regroupent sur une seule puce des composants auparavant assemblés sous forme discrète sur une carte imprimée. Les principaux avantages des systèmes sur puce par rapport aux circuits imprimés sont, d'une part, la réduction des coûts de fabrication d'un système complexe produit en masse et, d'autre part, une amélioration de la capacité de traitement. Cependant, les contraintes physiques sont beaucoup plus nombreuses et strictes pour un système sur puce, ce qui rend leur conception complexe et coûteuse. En général, un système sur puce contient un processeur programmable, de la mémoire embarquée, des unités de traitement spécialisées ainsi qu'une infrastructure de communication (Chang et al., 1999). La présence du processeur implique la cohabitation dans le même système de fonctions logicielles et matérielles. Les communications externes sont réalisées par des composants analogiques et mixtes, alors que les communications internes transitent par un système de communication intégré. Des composants de type radiofréquence, électromécaniques ou optiques peuvent également faire partie des éléments intégrés.

L'hétérogénéité des systèmes sur puce pose un défi de conception et de vérification considérable (Nicolescu, 2002). Au coeur du défi se situent l'interconnexion des nombreux blocs de propriété intellectuelle réutilisés, d'origine et de nature différentes, sur lesquels repose une proportion croissante du design (environ 80% en 2005 et 95% en 2010, selon Dataquest 2000). Le développement des systèmes sur puce exige donc une approche holistique qui repose sur des méthodes cohérentes, permettant de travailler à différents niveaux d'abstraction et de vérifier efficacement un circuit.

En général, les méthodologies et techniques de conception qui répondent à ces besoins mettent l'accent sur le concept d'aspects orthogonaux du design. Un système

peut être décomposé selon des axes indépendants, ce qui facilite l'exploration des solutions possibles. (Keutzer et al., 2000) décrit l'orthogonalisation des aspects *fonction vs. architecture* et des aspects *communication vs. calcul.* L'outil-méthodologie VCC de Cadence (Virtual Component Codesign) constitue un exemple d'orthogonalisation des aspects du design. Les composants d'un système sont vues comme des unités fonctionnelles virtuelles (Virtual Component) que l'on associe à des unités architecturales, réalisant l'orthogonalisation des aspects fonction et architecture, semblable à l'approche proposée par l'environnement POLIS(Balarin et al., 1997). Cet outil réputé difficile à utiliser a été abandonné par Cadence vers 2002.

L'orthogonalisation des aspects *communications vs. fonctionnalité* est à la base de la conception par interfaces (*interface-based design*). Dans ce type de conception, un design est caractérisé avant tout par la spécification des communications. Cette spécification peut se faire à différents niveaux, comme par exemple au niveau comportemental. Un raffinement itératif faisant appel à des mécanismes de synthèse peut ensuite être utilisé afin d'optimiser et d'implanter les communications. La conception par interfaces doit permettre la résolution des communications entre les différents modèles de calculs présents dans un système hétérogène. (Rowson et Sangiovanni-Vincentelli, 1997) proposent une méthodologie de conception par interfaces basée sur le passage de jetons et sur le raffinement incrémental et hiérarchique des communications. L'environnement de co-design Ptolemy (Buck et al., 1992) étend le découplage entre les communications et les fonctions et propose un mécanisme de connexion universel entre une variété de modèles de calcul. Le concept d'acteur polymorphique est employé pour encapsuler les différents modèles de calcul, coordonner les événements et servir d'interface avec une infrastructure de communication commune.

Une méthodologie bien adaptée à l'envergure des systèmes sur puce est la conception par plateforme (*platform-based design*), telle que présentée dans (Chang et al.,

1999) et (Martin et Chang, 2003). Ce dernier ouvrage fournit une définition précise de la conception par plateforme : il s'agit d'une méthodologie visant à réduire le temps et le risque associé à la conception et la vérification d'un système sur puce par une réutilisation poussée de combinaisons d'IP matériels et logiciels. Cette méthodologie emploie une approche *meet-in-the-middle*, qui recherche un point de rencontre entre l'espace fonctionnel et l'espace architectural (Figure 1.2). Le raffinement itératif de l'application est donc guidé par les contraintes d'une architecture jusqu'à l'obtention des performances désirées.

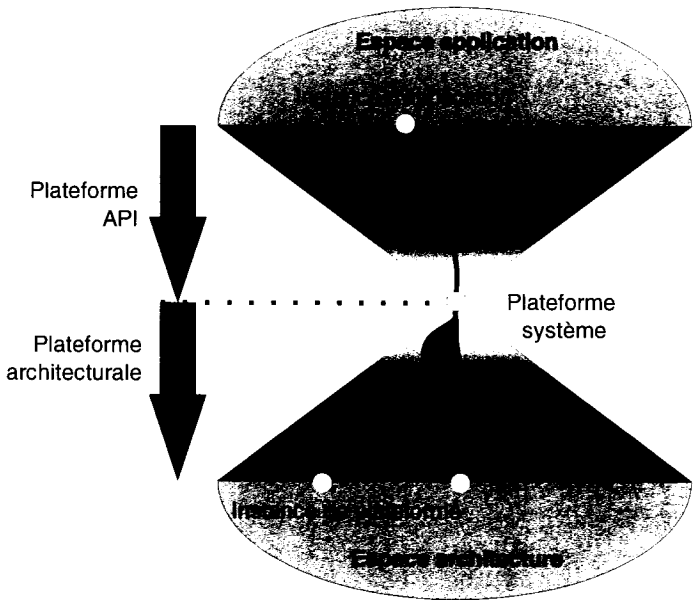

FIG. 1.2: L'approche meet-in-the-middle de la conception basée plateforme (Vincentelli, 2002)

La méthodologie repose sur une réutilisation systématique de blocs pré-existants et sur une hiérarchisation du design, le tout parfois relié à une plateforme matérielle flexible mais précise. Une plateforme typique contient une architecture de communication dont le but est de réduire la conception d'interfaces particulières. À un plus haut niveau d'abstraction, une plateforme offre un ensemble d'interfaces pré-caractérisées pour une famille d'applications cible (Sangiovanni-Vincentelli et Martin, 2001). Les systèmes conçus par l'approche basée plateforme sont générale-

ment des systèmes dérivés de systèmes existants ou des systèmes intégrant plusieurs fonctions auparavant séparées. Parmi les environnements de conception commerciaux et académiques basés sur l'approche plateforme, mentionnons Platform Express de Mentor Graphics[2], System Studio de Co-Centric [3], la famille de produits ConvergenSC de CoWare[4] ainsi que la plateforme SPACE de l'École Polytechnique de Montréal (Chevalier et al., 2003).

La conception par prototypage fonctionnel apparaît également comme une façon de maîtriser la complexité du design. En général, un prototype est une version primitive du système utilisée dans les premiers stages du design afin d'éclaircir certains aspects. (Kemp et al., 1996) décrit le rôle d'un prototype fonctionnel dans la validation et l'implantation finale d'un modèle complexe. Une approche moins formelle mais toujours axée sur la vérification est décrite dans (Lev et al., 2003). Le concept de prototype fonctionnel virtuel y est présenté comme véhicule unificateur d'un design hétérogène. Cette dernière approche a mené au développement de la plateforme de vérification *Incisive* de Cadence (Cadence, 2005).

1.2 Langages de conception pour systèmes sur puce

La pression de la productivité et le changement de paradigme vers des méthodologies holistiques ont mené au développement de langages de conception et de vérification plus expressifs et plus flexibles. La plupart de ces langages sont issus de la généralisation d'un langage de conception matérielle ou de la spécialisation d'un langage logiciel. Une catégorie à part, les langages de vérifications à haut niveau d'abstraction (*Hardware Verification Language*), ont été développés spécifiquement

[2]http://www.mentor.com/products/embedded_software/platform_baseddesign/
[3]http://www.synopsys.com/products/cocentric_studio/
[4]http://www.coware.com/products/convergensc.php

pour répondre aux besoins de vérification des systèmes complexes. D'aucuns ont proposé des méthodes et techniques issues de l'ingénierie du logiciel afin d'augmenter la productivité du design de systèmes électroniques, comme les techniques orientées objet (Kumar et al., 1994), ce qui a fortement influencé cette évolution des langages, tel que démontré ci-dessous.

SystemVerilog est un exemple de langage qui a évolué à partir d'un langage de description matérielle. Ce langage est essentiellement une extension de Verilog 2001, norme IEEE 1364-2001, auquel ont été ajoutés des mécanismes logiciels classiques et orientés objet, des mécanismes de vérification par assertions de même qu'une interface vers le langage C (Accellera, 2004). Ces mécanismes permettent une modélisation au niveau architectural plus concise d'un système matériel tout en conservant un chemin vers la synthèse.

D'une origine différente, SystemC n'en poursuit pas moins des buts très similaires à ceux de SystemVerilog. SystemC se compose d'une bibliothèque en C++ et d'un noyau de simulation. Il vise la modélisation de systèmes à des niveaux variables d'abstraction (OSCI, 2003). Le langage permet la modélisation de processus concurrents, d'opérations cadencées selon une horloge, de structures matérielles, etc. Reprenant le concept de Δ-cycle des simulateurs VHDL, le noyau de simulation exécute un modèle selon un ordonnancement par événements discrets basé sur une phase d'évaluation et une phase de mise à jour. SystemC ne permet pas la synthèse matérielle a priori, bien que plusieurs efforts soient en cours afin d'ajouter cet aspect au langage. Certains dépeignent SystemC et SystemVerilog comme deux langages compétiteurs[5], alors que d'autres les déclarent complémentaires, ciblant SystemC pour le design architectural de type descendant et SystemVerilog pour

[5]Observation recueillie au symposium *System-Level Design 2004 : Here And Now !* tenu à DAC 2004. Voir également l'article suivant : *EDA divided on SystemVerilog*, http://www.eetimes.com/issue/fp/OEG20030228S0021

le design matériel de type ascendant (Sciuto et al., 2004). Le temps nous dira si une spécialisation d'un des langages par rapport à l'autre aura lieu ou si les deux co-existeront dans une paix instable comme le font Verilog et VHDL. D'autres exemples de langages qui ont été utilisés pour la description de systèmes sur puce sont les langage Java (Helaihel et Olukotun, 1997) et C# (Lapalme et al., 2004).

À un plus haut niveau d'abstraction, deux langages graphiques sont utilisés dans la modélisation et le développement de systèmes-sur puces. Le premier, le SDL (Specification and Description Language)(International Telecommunication Union, 2000), est un langage adapté à la modélisation du contrôle et de la structure des systèmes de télécommunication. Il possède une sémantique bien définie, ce qui en fait un langage assez formel pour construire des modèles exécutable et pour être utilisé comme langage de programmation. (Bringmann et al., 1999) présente l'utilisation du SDL pour la spécification conjointe du matériel et du logiciel dans le cadre de prototypage rapide de système sur puce. (Daveau et al., 2002) présente une technique de génération de code VHDL à partir d'une spécification en SDL ; cette technique est centrée sur la synthèse des communications entre processus par le biais d'une représentation à plus haut niveau d'abstraction sous forme d'un canal abstrait. Parmi les outils qui utilisent le SDL comme langage de capture du design de systèmes embarqués, citons la suite Tau de Telelogc AB (Telelogic, 2005) et Cinderella SDL (Cinderella, 2005).

Le deuxième langage, UML (Unified Modeling Language), est un langage de modélisation versatile qui attire l'attention de la communauté système sur puce depuis quelques années. La version 1.1 du langage est la version initiale publiée par l'organisme responsable de sa normalisation, l'OMG (Object Management Group), en 1997. Né de l'intégration de plusieurs techniques de modélisation orientée objet, le langage avait comme but original de guider et documenter le design logiciel à l'aide de modèles à haut niveau d'abstraction. Son utilisation s'est rapidement étendue à

la modélisation de différents systèmes, comme les bases de données, les applications distribuées sur le Web et les systèmes temps réel (Booch et al., 1998).

Le vocabulaire d'UML est constitué d'objets graphiques, de relations et de diagrammes qui permettent de spécifier un système de façon structurelle et comportementale. Des règles sémantiques garantissent que les modèles sont cohérents et bien formés [6]. De plus, chaque élément de la notation graphique est supporté par une syntaxe et une sémantique particulière. Ainsi, une classe est représentée graphiquement par une icône rectangulaire, alors que les attributs, opérations et comportements qui la définissent sont détaillés à l'aide d'une spécification textuelle. Un modèle UML peut donc être raffiné itérativement à l'aide de cette spécification jusqu'au niveau de détail désiré.

Le langage UML repose sur la modélisation orientée objet, qui comprend des mécanismes relationnels comme l'héritage, l'encapsulation et l'association. Ces mécanismes permettent de construire efficacement le modèle d'un système complexe et ce de façon indépendante d'une implantation logicielle et matérielle. Bien qu'une implantation logicielle constitue le raffinement naturel d'un modèle UML, certains éléments peuvent être caractérisés par l'ajout d'information relative à une implantation matérielle (Martin et al., 2001). La répartition des éléments d'un modèle entre des processeurs logiciels et matériels peut elle-même être modélisée à l'aide de diagrammes de déploiement. Les mécanismes d'extension du UML, soient les stéréotypes, les marques (*tagged values*) et les contraintes, ont été utilisés pour la définition d'un profil de modélisation du temps, de la performance et de la planification (Object Management Group, 2002) pour le traitement en temps réel. Ce profil permet une modélisation précise du temps, des processus concurrents ainsi que de la qualité de service (QoS).

[6] (Henderson-Sellers, 2005) mentionne l'incohérence possible entre les événements modélisés dans un diagramme d'état et ceux modélisés dans un diagramme de séquence.

Indépendamment du profil mentionné ci-dessus, la version 2.0 du langage, entérinée par l'OMG en 2004, contient plusieurs nouveaux éléments particulièrement adaptés à la modélisation, au partitionnement et à l'analyse de systèmes sur puce. Ces nouveaux éléments ont d'abord été présentés dans des langages concurrents comme le SDL ou dans des techniques complémentaires comme ROOM. Cette dernière est une technique de modélisation orientée objet pour les systèmes temps réel, présentée dans (Selic et al., 1994) alors que le UML n'existait pas encore. La technique repose sur le concept de capsule, qui représente un élément architectural possédant un comportement et qui interagit avec son environnement par l'intermédiaire de ports (Selic et Rumbaugh, 1998). Un port est associé à un protocole, qui capture la sémantique des communications entre deux ports de capsules. Ce protocole établit en quelque sorte le contrat d'interface existant entre deux capsules. Le comportement d'une capsule est défini par une machine à états dont les transitions sont déclenchées par la réception de messages sur un port. Ces concepts de la technique ROOM ont trouvé en UML le véhicule d'expression idéal, ce qui a mené à une implantation particulière du langage connue sous le nom d'UML-RT (Rational Software Corporation, 2002). Cette implantation a été mise en oeuvre par Rational dans l'outil Rose-RT . Le concept de capsule a été intégré au UML 2.0 sous la dénomination *structured classifiers*(Object Management Group, 2004b).

De nombreux autres éléments ont été ajoutés au langage suite aux propositions de différentes origines. Par exemple, la représentation des contraintes de temps a été formellement ajoutée au diagramme de séquence, ce qui permet une modélisation précise des événements dans un système. Le diagramme de la Figure 1.3 illustre un échange de messages entre deux objets ainsi que plusieurs contraintes de temps spécifiées en langage OCL (*Object Constraint Language*, un langage complémentaire à UML). Une de ces contraintes indique que le délai vu par l'objet $User$ entre l'envoi du message $Code$ et la réception du message OK doit être compris entre d et $3*d$, où

d représente le temps de transmission du message *Code*. La modélisation du temps est également facilitée par l'ajout de diagrammes de temps (*Timing Diagram*), dont le but est d'illustrer le changement d'état d'un objet en fonction du temps. Ce diagramme constitue un format alternatif du diagramme de séquences et est similaire aux chronogrammes utilisés couramment en microélectronique.

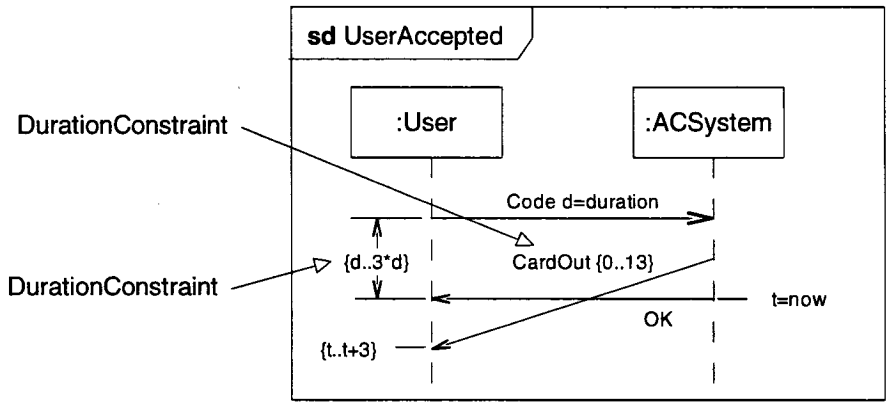

FIG. 1.3: Spécifications des contraintes de temps dans un diagramme de séquence (Object Management Group, 2004b)

En tant que langage, UML ne constitue pas une méthodologie de conception. Ainsi, (Lavagno et al., 2003) présente quelques méthodologies qui utilisent UML comme langage de spécification. Celle de (Chen et al., 2003), par exemple, est une méthodologie de conception par plateforme qui fait appel à un profil spécial pour la modélisation de plateformes de systèmes embarqués. Le profil comprend plusieurs stéréotypes (netlist, ressources, processus, etc.) qui permettent de modéliser une plateforme matérielle et ses interactions à différents niveaux d'abstraction. La méthodologie, basée sur Metropolis, est présentée à la Figure 1.4.

Mentionnons également les environnements de conception HaSOC (Hardware and Software Objects on Chip) (Edwards et Green, 2003) et SLOOP (System-Level design with Object-Oriented Process) (Zhu et al., 2002), tous deux basés sur UML. HaSoC est un environnement ciblé vers les systèmes sur puce qui propose d'effectuer une première modélisation orientée vers la sélection et la validation des fonctions.

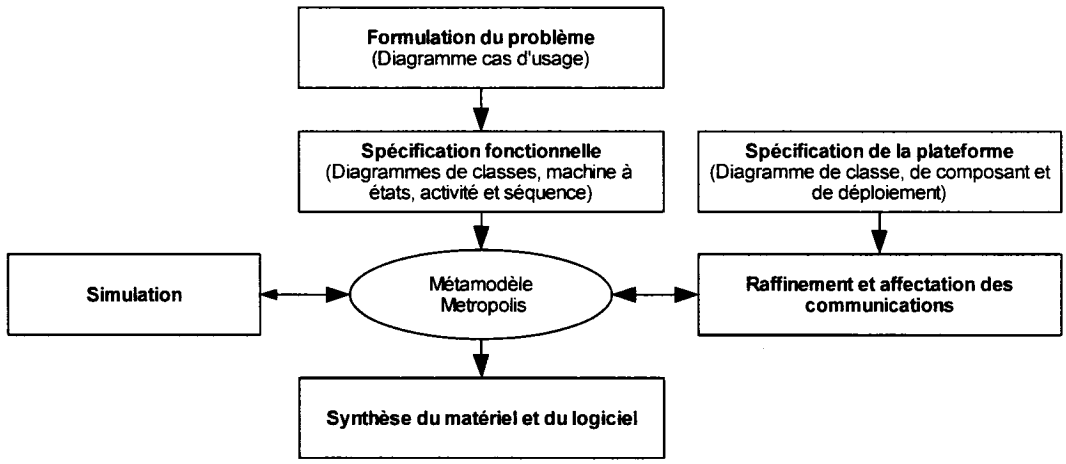

FIG. 1.4: Méthodologie de conception par plateforme basée sur UML (Chen et al., 2003)

Cette étape est suivie d'une modélisation ciblée où le partitionnement en fonction d'une plateforme matérielle, elle-même modélisée par un diagramme de composant, est effectué. Pour sa part, SLOOP est un processus de design mis au point aux laboratoires Fujitsu et qui utilise le UML, le C++ et le SystemC comme langages de spécifications et de modélisation. Le processus de design fait levier sur tous les mécanismes d'UML qui permettent la modélisation de systèmes temps réel et fait appel à SystemC pour l'évaluation de la performance d'un design.

La génération de code à partir des modèles constitue un élément clé dans l'augmentation de la productivité du design et constitue une technique importante dans l'approche MDA (Miller et Mukerji, 2003). Cette approche repose sur les principes de modèle indépendant d'une plateforme (PIM) et modèle spécifique à une plateforme (PSM). Bien qu'issue du monde logiciel, cette approche a été adaptée et proposée comme solution possible à l'automatisation du design, notamment par (Boulet et al., 2003).

Enfin, deux concepts issus du génie logiciel et qui peuvent être appliqués dans le cadre d'une méthodologie de conception de systèmes sur puce sont présentés. Il

s'agit des concepts de patron de conception (design patterns) et de contrat. Un patron de conception est un gabarit de solution pour une famille cible d'applications. Un patron est constitué de trois éléments : un problème commun, le contexte du problème et une solution générale. (Douglass, 2000) présente plusieurs patrons adaptés aux systèmes temps réel, dont les patrons conteneur, machine virtuelle et contrôle récursif. Quant au concept de contrat, il est utilisé afin de faciliter la réutilisation d'un composant logiciel par la formalisation de son interface et de son comportement. Ainsi, le contrat de base d'un composant peut-être représenté par l'ensemble des signatures de ses opérations, alors qu'un contrat de type comportemental spécifiera les pré- et post-conditions des mêmes opérations (Helm et al., 1990; Beugnard et al., 1999).

1.3 Normalisation des interfaces

Que ce soit pour faciliter la réutilisation ou la vérification, la normalisation des interfaces des modules IP est au coeur de nombreuses recherches. Reconnaissant l'importance et les défis de l'interconnexion des modules, un consortium de compagnies du secteur de la microélectronique ont formé la VSIA (Virtual Socket Interface Alliance) en 1996 (VSI Alliance, 2005). Cet organisme visait en premier lieu la normalisation des interfaces et des formats de représentation des modules IP matériels afin d'en faciliter l'échange. La norme ouverte Virtual Component Interface (VCI) fut le principal témoin de cet effort. Cette norme définit un protocole générique de communication point-à-point basé sur une plage d'adressage. Plus récemment, le rôle grandissant des modules IP logiciels dans la conception de systèmes sur puce a été pris en compte par la VSIA, de même que la nécessité de définir une méthodologie générale de réutilisation.

Une autre approche à la normalisation des interfaces de composants passe par la

définition d'une architecture de bus universel sur puce, dit médium partagé. À ce chapitre, plusieurs solutions ont été proposées, certaines ayant atteint un niveau d'acceptation important. CoreConnect d'IBM et AMBA de ARM sont certainement les deux architectures de bus les plus connues à l'heure actuelle. Ces architectures reposent sur la notion d'enveloppe (*wrapper*) afin d'interconnecter des composantes hétérogènes par le biais d'un bus commun et flexible, formant la base d'une plateforme de système sur puce. L'organisme Open Core Protocol, un consortium qui a récemment établi une alliance stratégique avec la VSIA (Wilson, 2003), propose également un protocole d'interconnexion de module IP sur la base de différentes interfaces, dont un protocole de bus similaire à AMBA appelé *Wishbone* (OpenCores, 2002). Il faut noter que l'ajout d'interfaces compatibles avec ce genre de spécification introduit un coût tant en surface qu'en temps.

L'interconnexion des composants qui constituent un système sur puce peut également se faire par le biais d'un réseau sur puce (Network on Chip). Ces micro-réseaux, inspirés des réseaux de télécommunication, réalisent les communications entre les différents composants d'un système sur puce par le transfert de paquets (Dally et Towles, 2001). Chaque composant client du réseau communique avec tous les autres clients et pas seulement ses voisins immédiats. La commutation et le routage des paquets sont réalisés de façon distribuée à travers la puce selon différents algorithmes (Benini et De Micheli, 2004). Les avantages des réseaux sur puce se situent au niveau de la structure, de la modularité et de la performance et sont acquis au prix d'une certaine surface de silicium. La plateforme StepNP de STMicroelectronics (Paulin et al., 2002), disponible sous format libre pour la recherche universitaire par le biais de la Société canadienne de Microélectronique, est basée sur une architecture d'interconnexion flexible implantée sous forme d'un réseau sur puce.

L'automatisation du design de circuits microélectroniques est une discipline nais-

sante où les techniques et méthodologies évoluent et se succèdent rapidement. Cette revue de littérature a tenté d'en saisir les principaux enjeux tels qu'ils se présentent en 2005. Les techniques et méthodologies poursuivront leur évolution, permettant l'intégration d'applications de plus en plus intelligentes dans des composants physiques de plus en plus petits et omniprésents.

CHAPITRE 2

ORTHOGONALISATION PAR LA SPÉCIFICATION D'INTERFACES

L'objectif principal d'une méthode de conception est d'améliorer la productivité du design tout en maintenant sa cohérence et sa continuité. Dans le contexte des systèmes sur puce, la conception basée plateforme assure la continuité du design en contraignant l'espace application en fonction de l'espace architecture (voir Figure 1.2). Cependant, les techniques qui permettent le passage entre les deux espaces sont ad-hoc et dépendent trop souvent de l'intuition des concepteurs. Le but de ce chapitre est de décrire comment la spécification des interfaces peut servir de pivot entre un modèle construit à haut niveau d'abstraction et son implantation. Un mécanisme de transaction est également introduit pour réaliser l'interface entre un modèle synchrone de type matériel et un modèle asynchrone de type logiciel.

2.1 Concepts

Un système microélectronique est un assemblage de modules qui coopèrent dans le but d'effectuer un traitement précis, défini par un ensemble de *requis*. Le fait de modifier la spécification de ce traitement requiert normalement la refonte de certains des modules. Cependant, l'impact d'une modification est difficilement contrôlable et peut résulter en une redéfinition complète du système si les modules qui le constituent ont été fortement couplés. Ceci est une conséquence des méthodes de conception centrées sur la fonctionnalité et la performance qui furent appliquées lors de la conception des premiers circuits intégrés complexes (Chang et al., 1999). La

réduction des coûts de communication et le respect des contraintes opérationnelles des composantes constituaient alors les principaux défis du design. L'envergure des systèmes sur puce actuels et la mise en valeur de la réutilisation ont déplacé la difficulté vers l'interconnexion efficace de composants hétérogènes à l'intérieur de contraintes physiques très strictes. Des techniques de conception qui séparent la fonctionnalité des communications ont été proposées afin de faciliter le développement de ces systèmes complexes. Qu'elles soient centrées sur la définition des interfaces, des communications ou des transactions, ces techniques accordent toutes une attention particulière aux communications inter-composants.

Le génie logiciel a réalisé la séparation entre la fonctionnalité et les communications par l'adoption du paradigme orienté objet. Un concept sous-jacent, l'encapsulation des détails d'implantation dans une interface, est particulièrement pertinent à la conception à haut niveau d'abstraction de composantes matérielles. L'encapsulation et la définition d'interfaces ont mené à la création d'infrastructures logicielles pour l'échange d'information, telle que CORBA (Object Management Group, 2004a), qui joue un rôle d'intermédiaire entre deux objets actifs peu importe leur implantation ou leur emplacement. De nombreux autres exemples, comme les bibliothèques de composants pour le développement rapide d'application (RAD), témoignent de l'importance d'encapsuler les détails d'implantation à l'intérieur d'interfaces bien définies. Au contraire du logiciel, les systèmes sur puce intègrent des composants variés tant par leur nature physique que fonctionnelle sans que les méthodes de conception n'offrent de mécanisme pour une représentation cohérente de leurs interfaces.

2.1.1 Interfaces

En se positionnant au niveau d'abstraction approprié, il est possible de spécifier les interfaces d'un système sur puce à l'aide des techniques utilisées en génie logiciel. En effet, un système peut être défini par une collection d'interfaces au travers desquelles l'information est transformée. La définition des interfaces d'un système est en quelque sorte une formalisation des requis de l'application qui sert de pivot entre l'espace application et l'espace architecture, ce qui offre les avantages suivants :

- Lors de l'ajout ou de la modification d'un requis, il est plus facile d'isoler et de modifier les parties affectées du système grâce à la modularité du design.
- Les interfaces d'un système donné constituent une infrastructure de communication qui peut servir de base pour des designs dérivés.
- La spécification des interfaces permet la construction de modèles basés sur les interactions entre composantes. Ceci est plus approprié à la modélisation des comportements concurrents que les approches algorithmiques traditionnelles (Motus, 2003).

La définition des interfaces est une technique qui se prête bien à la conception par plateforme où l'orthogonalisation des aspects est primordiale. La technique se caractérise par le fait de privilégier les communications (plutôt que le traitement) dans la définition d'une plateforme. On cherche donc à identifier les frontières naturelles à l'intérieur d'un système pour mieux en maîtriser la complexité. Des interfaces proprement spécifiées demeureront cohérentes lors du raffinement vers un niveau d'abstraction inférieur. Une plateforme peut donc être décrite par un ensemble d'interfaces pré-caractérisées pour un certain espace application.

Une interface doit décrire la fonctionnalité d'un composant de façon indépendante d'une implantation logicielle ou matérielle. Il convient d'abord d'identifier les dif-

férentes informations qui transitent par une interface, ce qu'illustre la Figure 2.1.

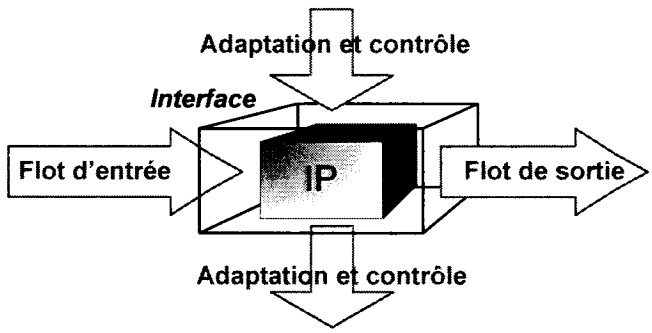

FIG. 2.1: Vue générale de l'interface d'un composant

Les flots de données d'entrée et de sortie représentent les données consommées et produite par le composant. Quant aux signaux d'adaptation et de contrôle, ils peuvent influencer le déroulement du traitement ou extraire des informations utiles à un autre composant. Cette caractérisation des informations permet une orthogonalisation encore plus fine des aspects, permettant la séparation du contrôle et du traitement ou la séparation de la configuration et de l'administration du composant. Selon l'approche ascendante, chaque composant est un élément d'un chemin de données ou de contrôle plus global. À l'inverse, selon l'approche descendante, un raffinement itératif divisera le composant en sous-composants significatifs.

Cette vue d'un composant se situe à un bas niveau d'abstraction. À un niveau plus élevé, une interface doit décrire les services offerts par un composant, toujours sans égards à son implantation. Ceci met au premier plan l'aspect *quoi* du composant, plutôt que le *comment*. Ces vues à différents niveaux d'abstraction peuvent être régie par le concept de contrat tel que défini et implanté par Bertand Meyer dans le langage Eiffel. D'après (Meyer, 1997),

> For a system of any significant size, the individual quality of the various elements involved is not enough. What will count most is the guarantee

> that for every interaction between two elements there is an explicit roster of mutual obligation and benefits, the contract.

Malgré l'attrait des méthodes orientées objet, leur utilisation pour la conception de composants matériels performants n'est pas triviale. En effet, l'ajout de couches d'adaptation et autres intermédiaires tels que ceux utilisés dans le monde logiciel se fait à un certain coût en terme de surface et de performance (Cyr et al., 2004). C'est d'ailleurs la principale critique envers les enveloppes standards pour noyaux réutilisables proposées par les groupes de normalisation comme la VSIA ou OCP/IP. Ainsi, le raffinement d'un système entre une description abstraite et une implantation doit se faire à l'intérieur d'un cadre dicté par les contraintes de performance, de coût et de temps d'accès au marché. Un ensemble d'interfaces pré-définies et ciblées vers une famille d'application peut guider la conception à l'intérieur de ce cadre. Il est utile, dans ce contexte, de sacrifier la généralité d'une plateforme de conception au profit de la performance des designs qui en sont issus.

En séparant les communications du traitement, une interface permet le découplage externe et interne des niveaux d'abstraction. Le découplage externe facilitera la réutilisation d'un module dans un contexte défini à un niveau d'abstraction différent du sien lors d'une conception ascendante. Le découplage interne permettra à différentes implantations associées à une même interface, donc considérées comme équivalentes, d'offrir des performances variées en fonction des besoins du design, de l'application et des techniques de déploiement. D'autre part, une interface peut encapsuler une implantation composite décrite à l'aide d'autres interfaces, permettant une hiérarchisation qui facilite le design.

2.1.2 Définition des contrats

Peu importe le niveau d'abstraction, les flots de données et les signaux de contrôle d'une interface ont des propriétés bien définies. Le concept de contrat peut être adapté aux besoins de la conception de systèmes sur puce et servir à la capture de ces propriétés. Un contrat spécifie les services, contraintes et requis qui doivent être respectés tant par le fournisseur que par le client. Quatre niveaux de contrats, dérivés de (Beugnard et al., 1999), sont définis dans le cadre de la conception de systèmes sur puce. Le contrat de base spécifie les mécanismes d'interaction par lesquels sont offerts les services du composant. Le contrat comportemental définit les règles d'utilisation des services en termes de pré- et post-conditions. Le contrat de synchronisation spécifie le comportement du composant dans un contexte où il interagit avec des processus concurrents. Enfin, le contrat de qualité de service quantifie le comportement attendu du composant. Chaque niveau de contrat sera illustré à l'aide d'un exemple à la section 2.4.

2.2 Méthodologie de conception

L'approche basée interface est une technique facilitant la mise en oeuvre d'une méthodologie basée plateforme. En effet, la définition des interfaces peut servir de gabarit lors du passage entre deux plates-formes définies à des niveaux d'abstraction différents. Cette section présente les détails de la méthodologie de conception par plateforme dans le cadre de laquelle sera utilisée la spécification d'interface. Cette méthodologie, essentielle au développement et à l'évaluation de la recherche, repose sur deux concepts de base : l'approche MDA et le prototypage fonctionnel.

2.2.1 Concept de base : l'approche MDA

La méthodologie repose sur un premier concept : le développement et la transformation itérative de modèles proposé par l'approche Model-Driven-Architecture (MDA) (Miller et Mukerji, 2003). Cette approche prévoit qu'un modèle indépendant de toute plateforme (PIM ou Platform-Independent Model) doit d'abord être développé afin d'être transformé en au moins un modèle spécifique à une plateforme (PSM ou Platform-Specific Model). L'avantage de cette approche est de favoriser le développement d'une spécification unique qui peut être réutilisée selon différents contextes d'application. Certaines parties d'un modèle PIM peuvent être transformées et spécialisées automatiquement à l'aide d'outils informatiques, rendant possible une certaine automatisation du design. Les principaux éléments de l'approche MDA sont illustrés à la Figure 2.2.

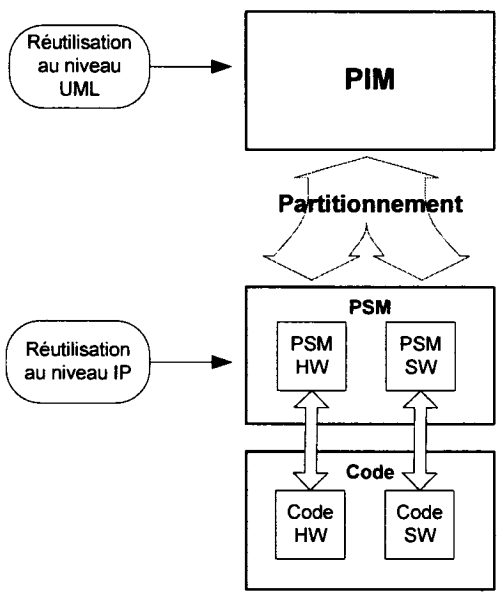

FIG. 2.2: Concepts fondamentaux de la méthodologie de conception utilisée

Dans ce contexte, une plateforme constitue un ensemble de ressources de traitement et/ou d'interconnexions décrites à un niveau d'abstraction qui peut varier. L'allocation des ressources d'une plateforme aux éléments du PIM se fait en fonction

des contraintes de performance et de design propres à l'application et constitue un PSM.

2.2.2 Concept de base : le prototypage fonctionnel

Le prototypage fonctionnel est le deuxième concept de base de la méthodologie utilisée. Le but du prototypage est de fournir une spécification cohérente et exécutable d'un système hétérogène afin d'en faciliter la validation et la vérification. L'hétérogénéité du système vient du fait que certains composants requièrent un cheminement de conception particulier, comme par exemple les composants analogiques et mixtes par rapport aux composants logiciels. D'autre part, selon l'application visée, des outils de conception spécialisés offriront une capture plus efficace des spécifications. C'est le cas d'environnements comme Simulink (The Mathworks, 2005) pour la modélisation de systèmes et de SPW (CoWare, 2005) pour la modélisation d'algorithmes de traitement du signal. Enfin, la présence de logiciel et de matériel à l'intérieur du même système est à elle seule source d'hétérogénéité.

Un prototype fonctionnel est construit à partir de la spécification textuelle de l'application par un travail concerté entre les ingénieurs système, les ingénieurs de vérification ainsi que des spécialistes de l'application. La spécification exécutable qui en résulte est un modèle qui est appelé à évoluer suite à un raffinement itératif. Un environnement de vérification fournit des données réalistes à traiter dès le début de la conception pour permettre l'observation et la validation du comportement dynamique du modèle. Le rôle premier de l'environnement de vérification est de représenter l'univers autour du système en développement. Il peut donc être construit de la façon la plus efficace possible à l'aide d'un langage de vérification matérielle, d'un langage de quatrième génération ou de composants IP de vérification. La spécification exécutable, l'environnement de vérification, ainsi que les

modèles de composants matériels sont mis en contexte dans la Figure 2.3.

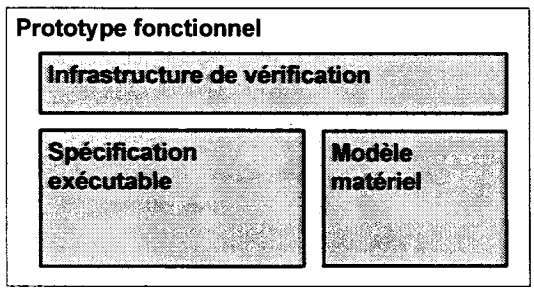

FIG. 2.3: Éléments du prototype fonctionnel

On remarque que l'utilisation d'une spécification exécutable fait apparaître tôt dans le design un besoin de vérification et de test. L'environnement de vérification servira dès les premiers stages du développement à valider le modèle ainsi que tout au long du développement afin d'effectuer la vérification détaillée de l'implantation. La spécification exécutable qui constitue le prototype fonctionnel étant en mesure de traiter sommairement les données qui lui sont fournies, la réutilisation du test de la spécification à l'implantation s'en trouve encouragée.

Une attention particulière doit être portée aux interfaces lors du prototypage fonctionnel. C'est en effet au niveau des interfaces entre les composants hétérogènes que la transmission et la traduction des signaux doit s'effectuer. Ceci concerne également les interfaces entre la spécification exécutable et l'environnement de vérification. Des mécanismes de transaction polyvalents et le moins envahissant possible, décrits à la section 2.5 de ce travail, feront du prototypage un cadre d'étude approprié à l'application de la conception basée interface.

2.2.3 Dynamique de la méthodologie

En appliquant le concept de MDA au prototypage, on obtient une méthodologie qui élargit l'utilité d'un prototype fonctionnel pour en faire un véhicule pour le

raffinement itératif. Une description comportementale, habituellement développée en logiciel, sert de point de départ au prototypage. L'utilisation de logiciel permet de construire une version exécutable de la spécification, qui peut ensuite être profilée et évaluée selon différentes métriques. Cette approche concorde avec le contexte actuel où le logiciel joue un rôle grandissant dans les systèmes sur puce.

Il faut cependant noter que l'utilisation de logiciel dans un système sur puce introduit une variance au niveau des temps d'exécution (Li et Malik, 1999). En effet, des techniques d'accélération de processeur comme le pipelinage, la prédiction de branchement et l'utilisation d'antémémoire offrent une performance qui peut se dégrader dans certaines situations (Hennessy et Patterson, 1996). Dans ce contexte, l'utilisation d'un simulateur de jeu d'instruction (Instruction-Set Simulator ou ISS) pour exécuter et profiler un segment logiciel peut être lente ou inexacte si l'engin de simulation ne tient pas compte des techniques d'accélération utilisées. Par opposition, l'exécution d'un segment de code sur un processeur réel permet un profilage exact et rapide d'un segment de code. Cette technique sera donc préférée à l'utilisation d'un ISS lorsque possible, même si elle se fait au prix d'une perte d'observabilité des signaux d'adressage, de données et d'interruption du processeur.

Le raffinement de la spécification exécutable se fait dans le cadre du prototype, qui constitue un véhicule hybride entre le modèle indépendant d'une plateforme et un modèle dépendant d'une plateforme. Le prototype peut donc contenir une proportion variable de composants indépendants décrits en logiciels et de composants spécialisés, spécifiques à une plateforme. Les composants spécialisés peuvent prendre la forme de modèles matériel synchronisés sur une horloge et décrits dans un modèle de calcul spécialisé (par exemple, le modèle *Synchronous Data Flow* utilisé pour le traitement numérique du signal). Ils sont introduits dans le prototype afin d'améliorer la performance de certaines portions du prototype ou pour réutiliser un composant IP existant. Le prototype atteint éventuellement un degré

de raffinement très près de l'implantation, qui fournit la performance requise en fonction de la plateforme d'implantation choisie.

Lors d'un raffinement itératif où plusieurs niveaux d'abstraction coexistent, les interfaces peuvent agir en tant que canaux d'information grâce à leur propriété de découplage. Un module en cours de raffinement peut donc être encapsulé dans une interface et intégré à la spécification globale du système, même si celle-ci est développée à un autre niveau d'abstraction. Ce concept est illustré à la Figure 2.4, où trois niveaux d'abstraction sont mis en contexte.

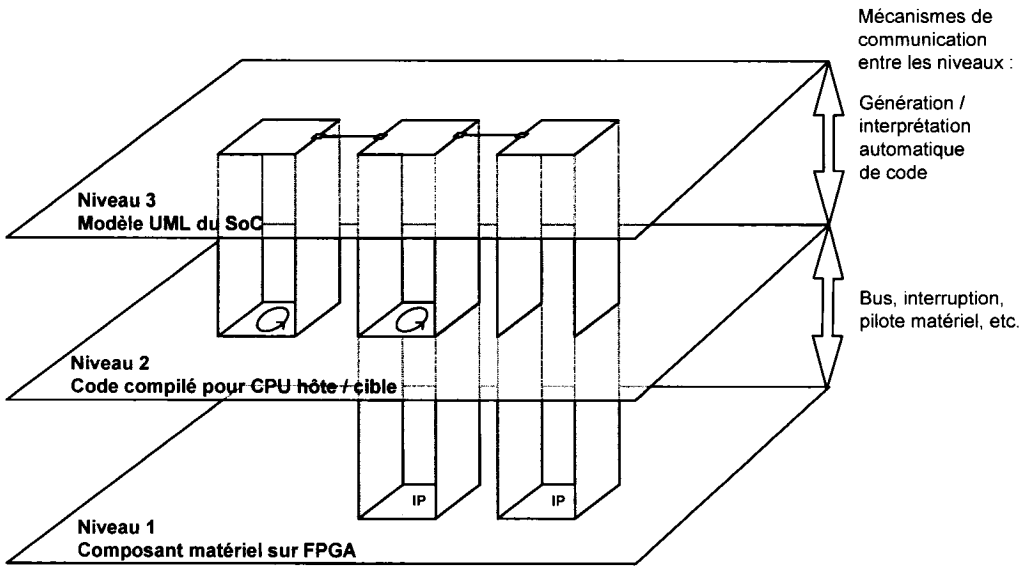

FIG. 2.4: Co-existence des niveaux d'abstraction

La principale difficulté dans l'interconnexion des interfaces est la cohabitation de composants décrits dans différents outils et modèles de calcul. Ainsi, certains composants modélisent du matériel et dépendent d'une horloge alors que d'autres composants sont décrits de façon purement logicielle et asynchrone. On obtient en fait un système globalement asynchrone et localement synchrone (GALS). Malgré cette hétérogénéité au niveau de la synchronisation, il est important de garder la spécification cohérente et exécutable tout au long du développement. Avant de présenter

les mécanismes d'interconnexion qui permettront cela, il convient de décrire le langage utilisé dans le développement et l'implantation de la spécification exécutable.

2.3 Utilisation d'UML comme langage de spécification et d'implantation

Le Unified Modeling Language (UML) offre les objets et diagrammes nécessaires à la construction d'une spécification de système sur puce non-ambiguë. Originalement un langage de modélisation pour le logiciel, UML a subi des modifications au cours des années qui ont étendu son champ d'application aux systèmes temps réel. Ainsi, la version 2.0 du langage, présentement en cours de finalisation, contient des éléments de modélisation de la structure et du comportement qui seront utilisés dans la présente recherche. L'utilisation de ces éléments permet la construction d'un modèle indépendant d'une implantation logicielle ou matérielle.

L'élément de base pour la modélisation de la structure est la *capsule*. Une capsule est une classe active spécialisée qui possède son propre fil d'exécution (thread). Ceci permet donc la modélisation de la concurrence, un aspect fondamental dans le développement de systèmes sur puce. Une capsule est définie par trois éléments :

- Des ports qui connectent une capsule à une autre capsule et qui permettent l'envoi et la réception de messages.
- Une machine à états qui définit son comportement.
- Une ou plusieurs sous-capsules qui en définit la structure interne.

Les ports définissent l'interface publique de la fonctionnalité d'une capsule. Ils sont l'unique moyen d'échange d'information entre des capsules collaboratrices. Un port est détaillé par un protocole et il peut donc servir à la définition du contrat d'interface d'une capsule.

Une capsule contient une machine à états privée qui implante son comportement. Il se peut également que le comportement d'une capsule soit réalisé par une ou plusieurs sous-capsule, auquel cas l'utilisation de la machine à états de la capsule principale est optionnelle. Certains aspects du comportement d'une capsule peuvent être rendus publics, en guise de documentation seulement, par le biais d'une machine à états associée à un port particulier ou d'un diagramme de séquence. Ceci permet la définition du contrat d'interface d'une capsule au niveau comportemental ou plus élevé. Quant à l'implantation détaillée d'une machine à états, elle peut être faite à l'aide de code C++ intégré dans les transitions entre les états. La Figure 2.5 illustre une capsule simple et sa machine à états.

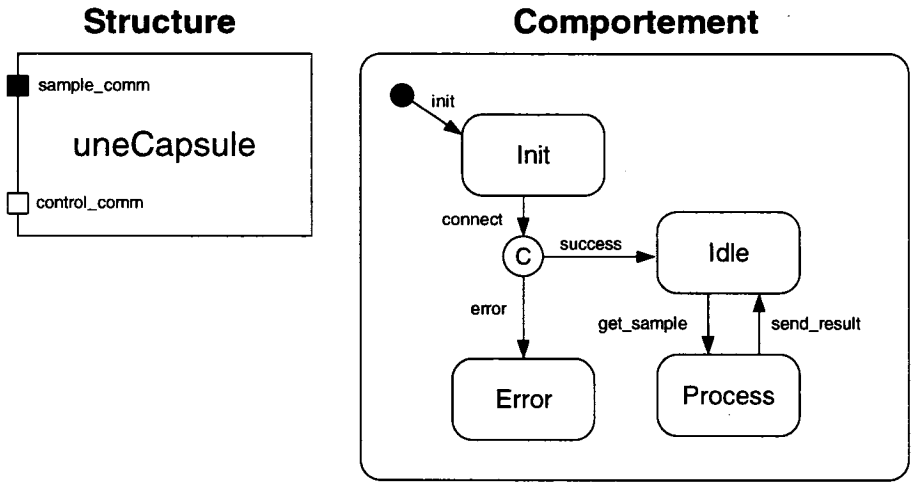

FIG. 2.5: Structure et comportement d'un composant modélisé en UML

Certains outils d'édition d'UML permettent la génération de code à partir des capsules et machines à états d'un modèle. Un modèle UML peut donc être défini indépendamment d'une plateforme pour ensuite subir une transformation automatique vers une plateforme spécifique. Cette capacité de génération est exploitée dans la présente recherche pour la production d'un code uniforme en C++. Le code détaillé des transitions sera automatiquement inséré au bon endroit dans le logiciel ainsi généré.

Il est aisé de voir comment une capsule peut modéliser une entité VHDL et ses ports, un SC_MODULE tel que défini en SystemC ou alors un processus logiciel indépendant. L'utilisation des mécanismes de modélisation orientée objet, tel que l'héritage et le polymorphisme, conjointement à la modélisation par capsule UML, permet une modélisation efficace et auto-documentée des principaux éléments fonctionnels d'un système sur puce.

La restriction des interactions aux ports uniquement crée une encapsulation forte du comportement et découple la capsule de son environnement. Ainsi, une capsule UML accompagnée d'un contrat d'interface bien documenté constitue une entité réutilisable qui concilie la portabilité d'une entité VHDL avec l'efficacité d'une approche orientée objet. Une telle capsule peut aussi être utilisée en tant que coquille vide dont l'implantation reste à définir, fournissant un moyen de spécification commun, autant pour des composants matériels que logiciels. Ce concept servira donc à la définition des interfaces du prototype, tel que présenté dans la prochaine section.

2.4 Spécification d'interfaces à l'aide d'UML

2.4.1 Requis fonctionnels applicables aux interfaces

Les interfaces doivent posséder certaines propriétés afin de guider efficacement la spécification et le développement d'un modèle UML. Trois requis sont énoncés : faciliter la réutilisation, permettre le raffinement itératif et permettre la validation et le profilage.

2.4.1.1 Interfaces pour la réutilisation

En accord avec le principe de réutilisation, le contenu structurel et comportemental d'une capsule peut être entièrement redéfini en autant que le contrat d'interface de la capsule soit respecté. Ceci est bien illustré par un diagramme structurel, où plusieurs instances de capsules collaborent entre elles (Figure 2.6). Dans un contexte structurel donné, chaque instance de capsule joue un rôle particulier, défini par les interactions avec son interface. Ainsi, peu importe quelle capsule est instanciée dans le rôle R1 de la Figure 2.6, seule compte la réalisation de l'interface qualifiée par ce rôle.

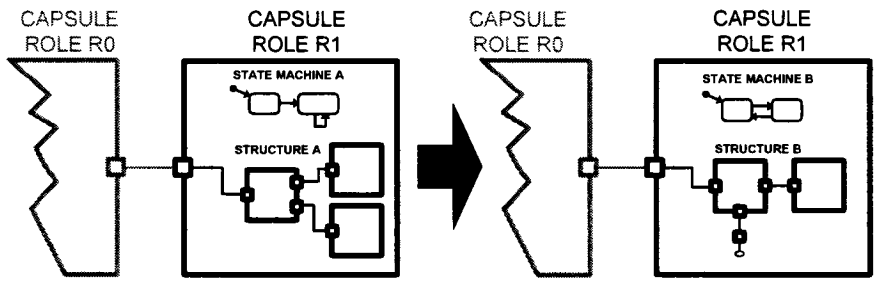

FIG. 2.6: Redéfinition du contenu d'une capsule de façon transparente

La méthodologie proposée encourage la réutilisation dès la spécification d'un nouveau module IP. L'interface de ce module IP est entièrement spécifiée à l'aide d'une capsule et de schémas UML qui documentent son contrat d'interface. De cette façon, le module peut être réutilisé plus facilement ailleurs, que ce soit dans des designs dérivés ou par un tiers parti.

Ceci peut être illustré à l'aide d'un exemple. Un élément générique dans la conception de systèmes sur puce est une source de données. Dans cet exemple, une source de données réutilisable et configurable est spécifiée sous forme de capsule. La source de données fournit des trames de longueur variable de mots de 32 bits. Ces données seront consommées par un utilisateur externe qui partage l'espace mémoire de la

source. Cet utilisateur doit au préalable fournir à la source de données l'adresse mémoire de base à laquelle les trames seront chargées. À la réception de ce paramètre, la source de données charge la trame en mémoire. Lorsque le chargement est terminé, la source de données émet un message d'acquittement et le processus recommence.

Les signaux échangés avec la source de données constituent le contrat de base de la capsule, qui est représenté par un protocole UML. Le contrat comportemental peut être défini par une machine à états où les transitions traduisent la séquence correcte d'échange des messages. Des contraintes de performance spécifiques peuvent s'ajouter à ces diagrammes pour définir le contrat de qualité de service. Dans le cas particulier de cet exemple, ce contrat peut-être simplement défini par le fait que la source doit fournir un débit de 10 Mbps avec un écart type tel que la performance minimale est de 9,5 Mbps. De plus, la latence et la taille des trames ne sont pas spécifiées. Quant au contrat de synchronisation, il serait nécessaire de le définir dans le cas où la source fournirait des données à plusieurs utilisateurs concurrents, un cas qui ne sera pas couvert dans cet exemple. La Figure 2.7 illustre le contrat comportemental et le contrat de qualité de service de la source de données.

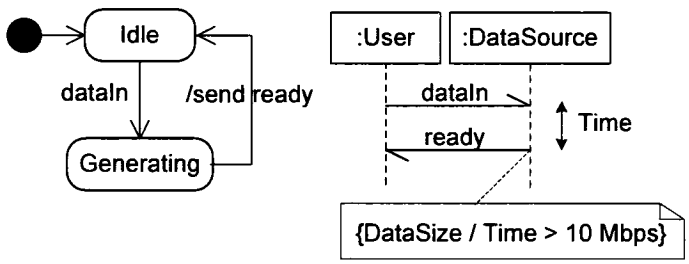

FIG. 2.7: Contrat comportemental et contrat de qualité de service pour une source de données générique

La source de données, documentée par son contrat d'interface, constitue un élément qui peut être facilement exporté vers d'autres designs. Sa définition est indépendante d'une implantation matérielle ou logicielle, ce qui permet d'utiliser la capsule

pour la construction d'un modèle de type PIM. Le processus de raffinement transformera cette coquille en un objet actif répondant aux contraintes d'une application particulière.

2.4.1.2 Interfaces pour le raffinement

Le raffinement est un processus qui amène les détails d'implantation d'un composant à un niveau de détail plus fin dans le contexte d'une spécification d'interface. Le niveau d'abstraction du composant s'en trouve abaissé, soit par la séparation en sous-composants ou par une implantation suivant une technologie donnée. Par exemple, les mécanismes internes d'une capsule peuvent être implantés à l'aide d'un modèle SystemC ou d'un module matériel sur un FPGA. Les ports de la capsule peuvent masquer la redirection des signaux vers un niveau d'abstraction arbitraire. Cette redirection peut aider au raffinement itératif d'un système, où chaque bloc est implanté au niveau d'abstraction le plus approprié de façon transparente.

Le raffinement concerne également la détermination des ressources allouées à l'exécution d'une fonction donnée. Une étape du raffinement pourrait être, par exemple, l'association entre une capsule et un processeur particulier. Ce genre de raffinement permet de mettre en lumière des résultats intéressants en ce qui concerne la parallélisation des processus et les coûts de communication qui en découlent. Un autre type de raffinement, celui des communications, peut également être appliqué sur les ports d'une capsule. Il n'y a pas de limite au niveau de détail que peut atteindre la définition du protocole d'un port donné. Une discipline de raffinement itératif peut donc être appliquée au long du développement afin d'effectuer le passage entre le niveau transactionnel et le niveau cycle.

L'exemple présenté ci-dessus peut-être réutilisé afin d'illustrer un cas de raffine-

ment. La génération de données peut s'effectuer de différentes façons qui doivent toutes satisfaire le contrat d'interface de la capsule. Par exemple, si la source de données est utilisée dans un processeur vidéo, elle pourrait faire appel à la carte vidéo afin de capturer les trames directement à partir d'une caméra. Dans d'autres cas, la signification des données n'a pas d'importance, tant qu'elles respectent le format prescrit. Dans ce cas, une source de données aléatoires peut être utilisée pour produire le flot de données. Le générateur aléatoire disponible dans l'environnement Matlab peut être utilisé à cette fin. Prenons par exemple la commande Matlab suivante :

```
d=int32(-2^15+rand(s,1)*((2^16)-1));
```

Cette commande permet la génération d'un tableau d de s mots de 32 bits. Le mécanisme de communication utilisé à l'interne par la capsule afin de gérer l'application Matlab est fourni par le système d'exploitation de la station de travail (*pipe* pour UNIX ou *COM Automation* pour Microsoft Windows). La technique de génération des données est complètement cachée du reste du modèle. Le *comment* a été réalisé à l'interne, alors que l'interface de la capsule ne spécifie que le *quoi*. Cependant, des informations utiles au raffinement restent à trouver, et ceci est le rôle du profilage.

2.4.1.3 Interfaces pour la validation et le profilage

Les ports des capsules se situent aux frontières naturelles d'un système et ils constituent un endroit privilégié pour la cueillette de données de profilage dynamique. Cette information découle des événements qui se produisent sur les signaux qui transitent par un port. Un événement se caractérise par l'endroit où il se produit, la nouvelle valeur qu'il annonce ainsi que le temps auquel il se produit. Une collection de ces temps peut être assemblée lors de l'exécution d'un système et utilisée

dans le profilage de certains aspects. Ceci permettrait, par exemple, de pondérer un temps global d'exécution entre les différents ports franchis lors de l'exécution, produisant un profil de charge.

L'exemple de la source de données peut être utilisé pour illustrer le profilage. Tel que mentionné, le débit d'une source de données constitue une caractéristique essentielle de même que la capacité d'une implantation donnée à respecter le contrat d'interface de la source. Par exemple, ce contrat stipulait simplement que le débit de la source doit être d'au moins 10 Mbps. Une implantation de la capsule peut alors être profilée afin de s'assurer que le contrat est respecté. Les résultats du profilage prendront tout leur sens dans un contexte d'application plus large tel qu'il sera présenté dans le chapitre des résultats.

Le fait de rendre disponible au niveau interface certains paramètres d'une implantation est une façon de garantir le respect des contraintes de performance. Par exemple, pour une implantation particulière de la source de données, un profilage peut révéler que la longueur de trame doit se trouver dans l'intervalle $[l_{min}, l_{max}]$ afin que la performance requise soit atteinte. Une caractérisation adéquate des interfaces assure que le comportement d'une capsule répond aux conditions du contrat d'un niveau d'abstraction à l'autre. Ainsi, un ensemble d'interfaces précaractérisées pour une classe particulière d'application est une façon de supporter une méthodologie basée plateforme.

2.4.2 Problème de l'hétérogénéité

L'hétérogénéité d'un prototype fonctionnel met en jeu la cohérence et le caractère exécutable de la spécification. Dans le cadre de ce travail, l'hétérogénéité se manifeste au niveau des interfaces entre éléments fonctionnels et affecte donc les

communications entre modules. À l'aide de spécifications d'interfaces en UML, il est possible d'aplanir les différences et d'abstraire la fonctionnalité des modules, peu importe leur nature, sous forme de capsules. C'est par son implantation qu'une capsule peut compromettre l'exécution, en introduisant une hétérogénéité dans les communications selon les deux aspects suivants :

- En terme de niveau d'abstraction. Une interface définie au niveau transactionnel ne peut interagir directement avec une interface définie au niveau cycle.
- En terme de modèle de calcul. Les différents modèles utilisés (machine à états, processus séquentiels communiquants, événements discrets, etc.) interagissent de façons fort différentes avec leur environnement.

Il est souhaitable de préserver cette hétérogénéité dans les communications pour plusieurs raisons. D'une part, ceci permet d'utiliser le niveau d'abstraction le plus approprié pour la description des éléments fonctionnels d'un prototype. D'autre part, l'effort de développement peut être distribué parmi différents spécialistes qui travaillent avec différents outils.

La présente recherche se concentre sur la résolution de l'hétérogénéité en terme de modèles de calcul. Un mécanisme de communication appelé *transacteur niveau modèle* (décrit à la section 2.5) est introduit pour effectuer le couplage des parties hétérogènes du prototype fonctionnel, et ce de façon transparente. Ainsi, un composant dont le comportement est décrit de façon asynchrone peut interagir avec un composant synchrone. D'un point de vue global, l'élément unificateur d'un prototype hétérogène est le réseau d'interfaces décrites à l'aide de capsules UML au niveau transactionnel. Certaines de ces capsules sont en fait des interfaces spécialisées dans la résolution des communications entre modèles de calculs : des transacteurs niveau modèle. De façon plus spécifique, nous cherchons à synchroniser un comportement décrit à haut niveau à l'aide d'une machine à états UML avec un comportement dé-

crit dans un simulateur à événements discrets, deux modèles de calcul couramment utilisés pour la spécification de composants logiciels et matériels.

Les machines à états UML décrivent le comportement d'une capsule et sont inspirées des statecharts de Harel (Harel, 1987). Ces machines à états ont les caractéristiques suivantes :

- Concurrence : plusieurs machines à états peuvent être définies et s'exécuter de façon concurrente dans différentes capsules.
- Réactivité : les transitions entre états sont effectuées lors de la réception d'un événement précis sur les ports de communication de la capsule.
- Hiérarchie : les machines à états peuvent être imbriquées afin de simplifier la modélisation de comportements plus complexes.

Bien que les machines à états UML aient plusieurs avantages pour modéliser les aspects dynamiques d'un système, quelques problèmes apparaissent lorsqu'il s'agit de communiquer avec un composant décrit dans un environnement de simulation par événements discrets. Par exemple, ces composants utilisent souvent des entrées / sorties registrées qui sont échantillonnées lors d'un coup d'horloge. Ce concept est étranger aux machines à états UML, avec lesquelles l'information est échangée par des messages asynchrones, plutôt que par des registres qui peuvent être échantillonnés. D'autre part, l'activation d'un signal par un composant simulé par événements discrets exige qu'un observateur soit présent afin de réagir aux changements d'état du signal (ceci définit la sensibilité d'un processus en VHDL, par exemple). En UML, ceci doit se traduire par l'envoi d'un message sur un des ports de la capsule cible.

L'infrastructure de vérification est normalement décrite à l'aide de composants simulés par événements discrets. Son rôle premier est d'agir comme le monde

extérieur, ce qui comprend, par exemple, l'émulation des processus analogiques-numériques qui se produisent aux frontières du système sur puce en développement. À plus petite échelle, l'infrastructure de test peut agir en tant qu'environnement numérique d'un sous-composant, jouant le rôle d'un coprocesseur synchrone disponible dans une bibliothèque de modules IP. Dans tous les cas, il est plausible que le monde extérieur au système en cours de développement se manifeste sous forme d'échantillons synchrones. Afin de permettre la communication entre les deux mondes, les entrées / sorties des composantes synchrone doivent être traduites sous forme de messages qui seront interprétés par les machines à états UML. Ceci est précisément le rôle des transacteurs, qui sont décrits en détails à la prochaine section.

2.5 Le rôle des transacteurs niveau modèle

Jusqu'à maintenant, nous avons présenté l'aspect extérieur d'une interface, caractérisée par un contrat et une encapsulation forte des détails d'implantation. Certaines interfaces vont jouer un rôle de médiateur entre la spécification exécutable, décrite en UML, et des composants spécialisés, décrits dans des simulateurs par événements discrets. Cette section présente les détails d'implantation de ces interfaces spécialisées appelées *transacteurs niveau modèle.*

2.5.1 Situation par rapport à la modélisation au niveau transactionnel

Pour débuter, il importe de bien définir le concept de transacteur niveau modèle et comment il se positionne par rapport aux autres formes de transactions. Un transacteur niveau modèle est une interface spécialisée qui établit un lien de communication entre un composant simulé par événements discrets et un composant

exécuté en temps réel, et ce dans le contexte d'une spécification exécutable décrite à l'aide de capsules UML. La Figure 2.8 situe les transacteurs par rapport aux trois éléments du prototype, à savoir l'infrastructure de test, le modèle UML exécutable, ainsi que les modèles de composants matériels.

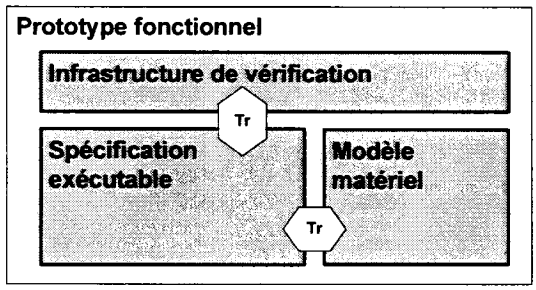

FIG. 2.8: Utilisation des transacteurs dans le prototype fonctionnel

En SystemC, un transacteur est utilisé pour la modélisation et la vérification au niveau transactionnel. Ce transacteur permet la communication entre une interface définie au niveau transactionnel et une interface définie à un niveau d'abstraction habituellement moins élevé (le niveau signal par exemple, tel que décrit dans (Alquier et al., 2003)) . Cette utilisation des transacteurs est biaisée vers le matériel, comme le sont habituellement les modèles développés en SystemC. De plus, ces transacteurs dépendent de l'ordonnanceur SystemC qui contrôle le temps et l'ordre d'exécution des processus (OSCI, 2003).

Le concept de transacteur niveau modèle, proposé dans le présent travail, est une entité à haut niveau d'abstraction qui contient un mécanisme permettant de synchroniser des événements entre le domaine simulé par événements discrets et le domaine temps réel. Dans le domaine dit temps réel, la fonctionnalité du composant est mise en oeuvre en l'absence d'un simulateur. Dans ce cas, un transacteur gère les transactions entre ce composant exécuté en temps réel (sur un processeur hôte ou un processeur cible) et un composant exécuté dans un simulateur à évé-

nements discrets. Ainsi, un prototype fonctionnel équipé de ces transacteurs est soumis à une *co-exécution hétérogène* plutôt qu'à une *co-simulation.*

2.5.2 Modèle client-serveur et synchronisation

Un serveur est une entité qui répond à des requêtes clients. De façon similaire, une application intégrée peut être vue comme une entité autonome qui offre un service de traitement de données à des clients. Si l'entité de traitement cesse de fonctionner, les requêtes clients rebondissent tout simplement. Tandis qu'en l'absence de requêtes clients, le serveur entre en mode de veille ou effectue quelques tâches internes. Dans le contexte du prototypage fonctionnel, la spécification exécutable se présente comme le serveur et l'infrastructure de vérification comme le client. D'un point de vue raffinement matériel, la spécification exécutable devient client de coprocesseurs matériels qui offrent une accélération de certains traitements.

Le transacteur qui établit la communication entre les composants hétérogènes du prototype est composé de deux moitiés, une qui fait office de serveur et l'autre de client. Chacune de ces moitiés est le miroir de l'autre et elles « existent » toute deux dans un modèle. Par exemple, lors de la communication entre une infrastructure de test simulée par événements discrets et la spécification exécutable, la moitié serveur est un élément du modèle UML alors que la moitié client est un élément de l'infrastructure de test. Chaque moitié est à son tour constituée d'une interface frontale, compatible avec le modèle où le transacteur se trouve, et d'un support arrière, qui implante la signalisation client-serveur.

La synchronisation des événements entre le monde simulé par événements discrets et le monde exécuté représenté par les machines à états UML repose sur l'identification des éléments de base du traitement. Ces éléments de base proviennent du

monde extérieur et sont émulés par l'infrastructure de test. C'est donc celle-ci qui détermine l'horloge maître du prototype en définissant et exportant les limites d'un cycle de base aux portions asynchrones et synchrones du prototype (cette méthode est une application des concepts présentés dans (Jantsch, 2005)). Dans un processeur vidéo par exemple, un cycle de base peut être défini par le traitement d'une image, d'une ligne de l'image ou d'un pixel. Dans un système de traitement numérique du signal (DSP), un cycle de base peut être le filtrage d'un échantillon du signal d'entrée. Plus généralement, un cycle de base se compose d'une chaîne d'événements prévisibles et répétitifs. Ainsi, l'échange d'un message entre un composant simulé et une capsule UML est rendu possible soit parce que le moment exact où il se produit est connu et attendu, soit parce qu'un mécanisme dédié observe l'état du signal avant de déclencher une suite prédéterminée d'actions.

Afin de simplifier le prototypage, la synchronisation repose uniquement sur les relations de causalité entre les processus et seule la réception des événements déclenche l'exécution d'un processus. Selon le principe de parfaite synchronie (Benveniste et Berry, 1991), aucune notion de temps physique n'est introduite dans le prototype, ce qui permet une concentration des efforts de conception initiaux sur l'aspect fonctionnel du système. Ainsi, les délais de propagation dans les circuits numériques modélisés ne sont pas pris en compte et l'exécution de fonctions par les machines à états prend un temps nul. Les événements qui se produisent dans le prototype sont linéarisés selon un principe de causalité transitive, ce qui permet un ordonnancement partiel suffisant pour une exécution fonctionnelle. Globalement, seule la complétion d'un cycle de base peut faire avancer « l'horloge » du système, qui se trouve être une horloge logique. Les horloges des processus synchrones, utilisées lors du raffinement du prototype, sont locales et indépendantes de l'horloge logique globale. La communication entre le monde simulé par événements discrets et le monde exécuté est réalisé par un mécanisme de production-consommation de

jetons : les processus sont déclenchés conditionnellement à la réception d'un jeton et produisent eux-mêmes les jetons requis par les processus subséquents.

2.5.3 Implantation à l'aide de sockets

Les sockets Berkeley sont un mécanisme efficace et polyvalent qui permettent à deux composants logiciels de partager des données. Ils peuvent être utilisés pour l'implantation du support arrière des transacteurs. Du côté client comme du côté serveur, le mécanisme de socket est encapsulé dans un élément de modélisation compatible avec le langage utilisé (pour UML, cet élément est une capsule). D'autres mécanismes sont disponibles pour l'implantation de cette portion des transacteurs, mais leur contraintes d'utilisation sont plutôt sévères. L'utilisation de mémoire partagée, par exemple, est un mécanisme de communication inter-processus très performant, mais qui requiert que les processus aient accès à une plage commune d'adressage. Si les processus sont exécutés sur différents processeurs, ceci exigera normalement l'utilisation d'une architecture à mémoire partagée distribuée (*Distributed Shared Memory*). Cette architecture complexe ajouterait au prototype des mécanismes qui ne font souvent pas partie du système. Un autre mécanisme de communication qui pourrait être utilisé est celui des fichiers partagés. Malgré sa grande simplicité, ce mécanisme requiert la création d'un système de fichiers et peut exiger la présence d'un système d'exploitation pour le supporter. Enfin, d'autres mécanismes comme CORBA et COM sont aussi disponibles, mais ils offrent une couche d'abstraction supplémentaire par rapport aux sockets qui n'est pas nécessaire dans le cadre de ce travail.

Les sockets peuvent être utilisés de concert avec les protocoles TCP et IP pour établir une liaison point-à-point fiable basée sur de simples opérations de lecture et d'écriture. La communication entre la spécification exécutable décrite en UML

et les composants simulés par événements discrets peut être synchronisée à l'aide de lectures bloquantes (attente et réception d'un jeton) et du tampon d'écriture (envoi d'un jeton). Ainsi, une opération de lecture fonctionne de façon similaire à une bascule ou un registre FIFO, ce qui permet une synchronisation fine de la machine à états avec le monde matériel.

La performance du mécanisme des sockets dépend de trois facteurs : la vitesse des processeurs engagés dans le lien de communication, les caractéristiques du medium de communication utilisé et la taille des paquets de données envoyés. La performance nominale d'un lien de communication peut être mesurée à l'aide de l'outil *Iperf*[1]. Ainsi, pour une configuration en mode de retour de boucle (loopback) sur un processeur Pentium cadencé à 1.4 GHz, l'outil indique un débit de 3.35 Gbps. Lorsque le lien de communication se fait par le biais du protocole Ethernet, le débit maximal est habituellement dicté par la plus lente des interfaces réseau impliquées dans la communication : 10 Mbps, 100 Mbps et 1 Gbps sont les débits les plus courants. Cependant, les ressources de traitement requises par un socket TCP/IP sont élevées et le transacteur devrait pouvoir être facilement retiré d'un prototype en cours de raffinement, afin d'en améliorer la performance. En effet, le fait qu'une partie du prototype soit exécutée en temps réel met sa performance à la merci de tout processus qui partage les ressources de traitement. Une performance médiocre du prototype due aux communications pourrait même justifier le retour à l'utilisation d'une forme d'ISS.

[1]La documentation de *Iperf* est disponible à l'adresse suivante : http://dast.nlanr.net/Projects/Iperf/

CHAPITRE 3

APPLICATION ET RÉSULTATS DE LA SPÉCIFICATION D'INTERFACES EN UML

Le UML a été présenté comme un langage adapté à la modélisation structurelle et comportementale d'un système et de ses composants. Il est légitime de se demander jusqu'où peut aller le raffinement des modèles dans la progression du design. Bien qu'UML permette un niveau de détail très précis, son utilisation doit se limiter à ce qui fait sa véritable force : la modélisation à haut niveau d'abstraction. Ce chapitre présente l'utilisation d'UML pour la spécification des interfaces et du comportement d'une application de radio réalisée par logiciel (RRL). La mise en oeuvre de la méthodologie décrite au chapitre précédent a permis le développement et la caractérisation de plates-formes à différents niveaux d'abstraction. Ensemble, ces plates-formes serviront de base pour le développement rapide de designs dérivés pour des applications de RRL.

Les principaux termes utilisés dans ce chapitre sont définis dans le tableau 3.1.

3.1 Contexte de l'application

L'utilisation d'une méthodologie de conception basée plateforme se fait dans le contexte d'une famille d'application. En effet, la définition et l'utilisation des éléments pré-contraints d'une plateforme doit tenir compte d'un flot de données ayant des caractéristiques particulières. De plus, les plateformes décrites à un plus bas niveau d'abstraction doivent intégrer les contraintes de performance physique de l'ap-

TAB. 3.1: Définition des termes utilisés

Terme	Définition	Exemple
Modèle	Représentation à un haut niveau d'abstraction des fonctions et de l'architecture d'un système	Machine à états UML, diagramme de classe
Spécification exécutable	Modèle auquel du code exécutable est ajouté	Capsule UML avec machine à états contenant du code C++
Prototype fonctionnel	Ensemble cohérent et exécutable des modules d'un système et d'une infrastructure de test	Environnement de test Simulink et unités sous test écrites en UML
Plateforme	Ensemble de composants pré-conçus et pré-vérifiés qui supportent une famille d'applications	Plateforme pour applications RRL
Plateforme matérielle	Équipement électronique configurable dédié au prototypage de systèmes mixtes logiciel / matériel	Plateforme Altera Stratix Development Kit.
Outil	Logiciel qui permet la manipulation (édition, compilation, simulation, gestion de fichiers, etc.) d'un langage de conception	Compilateur *GNU gcc*, simulateur *Cadence ncsim*, éditeur d'UML Rational Rose
Méthodologie	Ensemble des procédures recommandées dans l'exécution de tâches de conception	Flot de conception recommandé par la Société canadienne de microélectronique
Langage	Vocabulaire et règles qui représentent un domaine de conception particulier.	Matlab, VHDL, C++, UML

plication cible. Les deux exemples suivants illustrent bien cette relation application-plateforme :

- Un système de traitement vidéo de qualité HDTV possède d'importantes mémoires tampon pour l'application des algorithmes à une image complète et doit fournir un débit binaire entre 20 Mb/s et 40 Mb/s.
- Un récepteur RRL par contre ne possède pas ou peu de ces mémoires mais peut devoir traiter des débits variables selon l'étape du traitement, atteignant l'ordre du GHz, près de l'interface radio, en descendant jusqu'au kHz, près de l'interface utilisateur.

Le cadre du présent travail est défini par une application RRL pour transmission sans fil de données numériques. La RRL est une architecture pour systèmes de communication sans fil dont les couches physique et de liaison peuvent être configurées - ou réalisées - par du logiciel (Mitola, 2000). Le principe de la RRL est de positionner un convertisseur analogique-numérique le plus près possible de l'antenne afin de réaliser un maximum de fonctions à l'aide d'un logiciel de traitement numérique du signal. Ceci procure une flexibilité accrue à la radio, qui peut être reprogrammée afin de supporter différentes modulations, encodages et protocoles. La Figure 3.1 présente un diagramme simplifié d'un récepteur RRL. Le signal reçu par l'antenne est numérisé à l'aide d'un convertisseur analogique - numérique immédiatement après amplification par le contrôle automatique du gain (AGC). Ensuite, le bloc de traitement numérique du signal effectue les traitements de démodulation, décodage, filtrage et analyse. Ce bloc peut être implanté sur un processeur de type DSP, sur un processeur générique, sur un FPGA, etc. Les ressources d'implantation sont habituellement fournies par une plateforme matérielle plus ou moins spécialisée. Enfin, un convertisseur numérique - analogique peut être introduit afin de terminer la chaîne de traitement. La conception d'un système sur puce pour ce genre d'application réunit des composantes logicielles et matérielles et requiert une allocation

équilibrée des ressources pour un maximum de flexibilité.

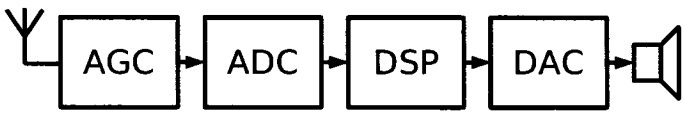

FIG. 3.1: Diagramme bloc simplifié d'un récepteur RRL

Un des composants clés d'un récepteur radio est l'égaliseur adaptatif. L'inclusion de ce composant dans la chaîne de traitement permet de compenser les distortions de phase et d'amplitude engendrées par le canal de transmission. Les trois principales architectures d'égaliseur adaptatif utilisées aujourd'hui sont l'égaliseur adaptatif linéaire, l'égaliseur adaptatif à retour de décision et l'égaliseur adaptatif à annulation d'interférence (Proakis, 2000; Gersho et Lim, 1981). Ces égaliseurs sont basés sur un filtre numérique à réponse finie (FIR) dont les coefficients sont mis à jour selon un algorithme récursif qui peut être affiné en fonction de la réponse impulsionnelle du canal de transmission (Qureshi, 1985). Dans le cadre de ce travail, nous allons développer un égaliseur adaptatif linéaire avec une mise à jour des coefficients selon l'algorithme des moindres carrés. La mise à jour est basée sur l'équation suivante :

$$C_{k+1} = C_k + \Delta \epsilon_k V_k^* \tag{3.1}$$

Le vecteur C_k représente les coefficients à l'itération k, ϵ_k est l'erreur sur le signal au temps $t = kT$, V_k^* est le vecteur des échantillons reçus et Δ est une valeur positive choisie afin d'assurer la convergence de l'algorithme. Cet algorithme est représenté à la Figure 3.2, extraite de (Proakis, 2000).

L'égaliseur adaptatif est représentatif d'un système sur puce hétérogène car certains traitements peuvent être réalisés en logiciel ou en matériel, selon les contraintes de performance requises. Une approche holistique est donc requise pour la conception d'un égaliseur pour RRL.

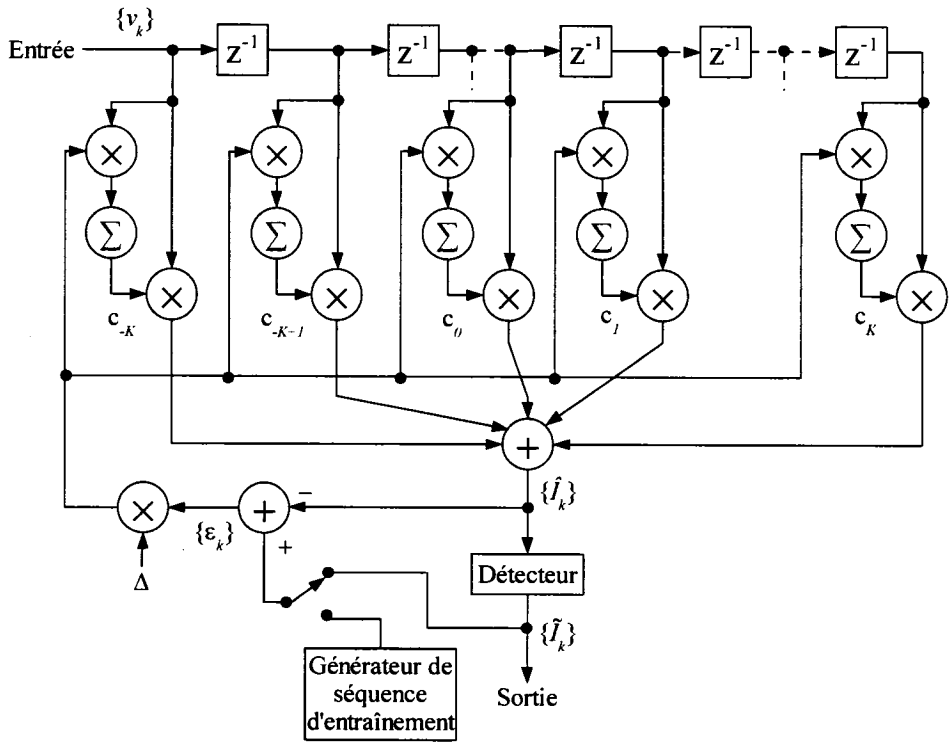

FIG. 3.2: Égaliseur adaptatif linéaire basé sur l'algorithme LMS

3.2 Construction du modèle PIM

La conception de l'égaliseur débute par la construction d'un modèle indépendant d'une plateforme, le modèle PIM. Le UML est utilisé à cette fin pour capturer la spécification du système et de son environnement opérationnel à partir des spécifications textuelles. Afin de bénéficier de la facilité d'édition et de la génération automatique du code à partir des modèles, un éditeur UML doit être utilisé. Le choix s'est arrêté sur l'outil *Rational Rose Real-Time* (Rational Software Corporation, 2003) pour sa disponibilité auprès de la communauté de recherche académique, sa compatibilité avec les éléments de modélisation d'UML 2.0 tels que les capsules, ports et protocoles et sa capacité de génération de code C/C++ à partir du modèle.

La chaîne de traitement d'un récepteur RRL est analysée afin que ses frontières

naturelles soient transposées sous forme de capsules UML. Le diagramme structurel résultant est illustré à la Figure 3.3.

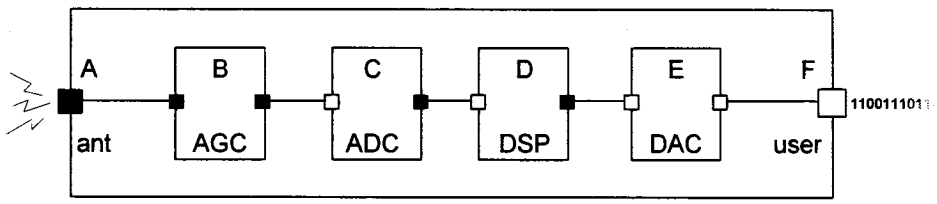

FIG. 3.3: Diagramme structurel d'un récepteur RRL

À ce stade, le modèle PIM constitue une représentation structurelle non exécutable du prototype. Des comportements doivent être insérés à l'intérieur des interfaces afin de créer une version exécutable et vérifiable. Une première orthogonalisation est faite en séparant les capsules qui font partie de l'environnement opérationnel de celles qui font partie du système sous développement (l'égaliseur). L'environnement opérationnel peut alors être modélisé dans un outil séparé et augmenté de mécanismes d'analyse pour servir d'environnement de vérification. Dans le cas d'une application de RRL, l'outil tout désigné pour cette fin est Simulink de Mathworks (The Mathworks, 2005). Il est intéressant de noter que les parties du modèle de la Figure 3.3 appelées à être raffinées à l'aide de Simulink peuvent faire l'objet d'une première transformation automatique. En effet, une capsule UML correspond au niveau structurel à un bloc *sous-système* de Simulink. De même, les protocoles qui définissent les ports de la capsule peuvent être éclatés en leur signaux constituants qui seront ensuite utilisés pour relier les sous-systèmes Simulink. Un script qui analyse le modèle UML et le transforme en un ensemble de sous-systèmes Simulink suffit à cette tâche (voir annexe III). Les sous-systèmes sont ensuite raffinés manuellement à l'aide des différentes bibliothèques de Simulink. Le modèle qui en résulte est présenté à la Figure 3.4.

Ce modèle est construit pour générer et analyser les données de radio numérique qui seront traitées par l'égaliseur. Plusieurs composants prédéfinis sont disponibles

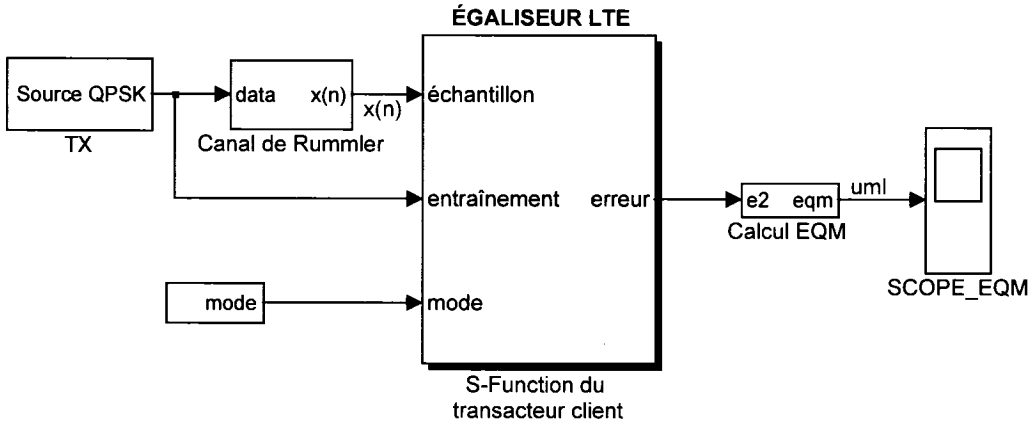

FIG. 3.4: Modèle Simulink de l'environnement de vérification

pour ce faire, comme un bloc d'encodage QPSK, des modèles de canaux radio et des blocs d'analyse des signaux et d'affichage.

Quant au système en développement, son comportement est raffiné en UML selon la méthode MDA afin d'établir un chemin efficace vers une implantation logicielle, matérielle ou mixte de l'égaliseur. La capsule principale, qui représente l'égaliseur, est raffinée à l'aide de sous-capsules de contrôle, de filtrage, de calcul d'erreur, de mise à jour des coefficients et autres traitements connexes (FIFO, multiplexeurs, etc.). Le diagramme structurel de l'égaliseur est illustré à la Figure 3.5.

Une des capsules clés est la capsule de mise à jour des coefficients de l'égaliseur, qui se fait selon l'équation 3.1. Une machine à états est créée pour la capsule de mise à jour des coefficients afin d'établir le comportement de la capsule (Figure 3.6). L'équation de mise à jour est implantée en code C++ dans la transition `updating` de la machine à états (Listage 3.1).

Il est intéressant de constater que la machine à états est presque banale. Dans un système dominé par les données tel qu'un égaliseur, la majorité du travail se produit lors de la réception périodique de la donnée à traiter, qui se modélise par un seul couple état - transition. L'utilisation de cette représentation n'apporte donc

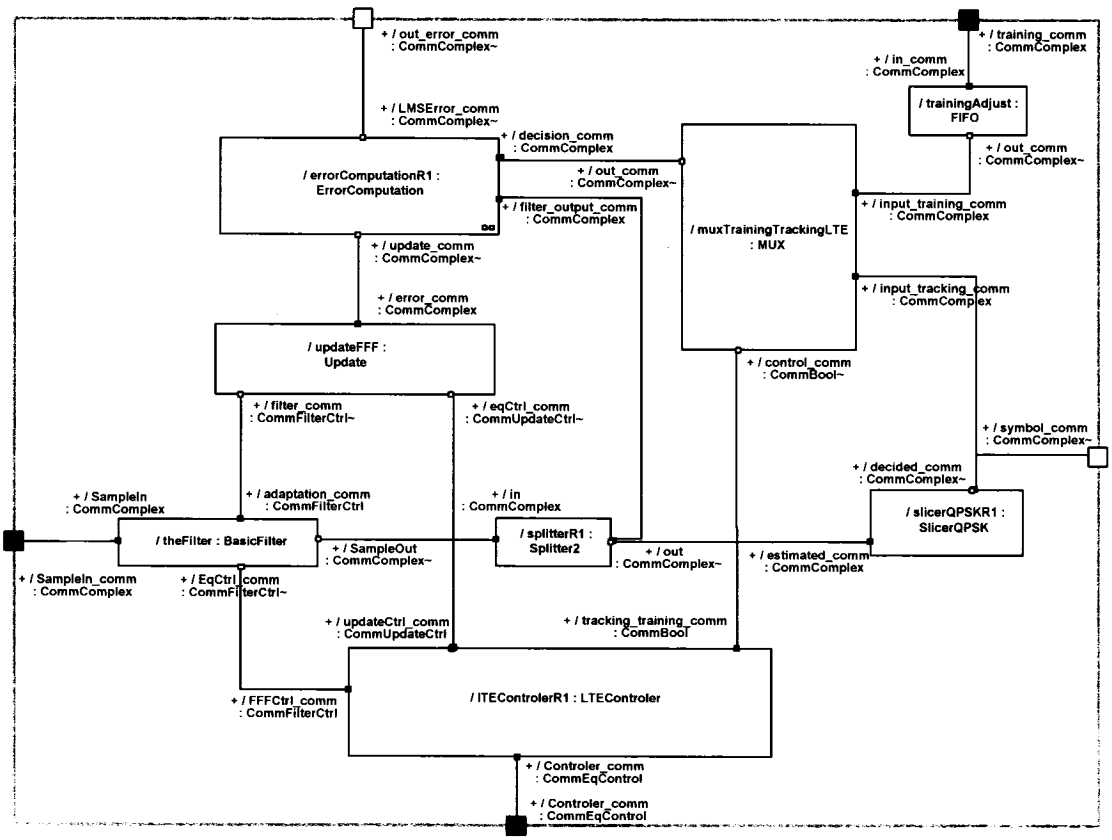

FIG. 3.5: Diagramme structurel d'un égaliseur adaptatif linéaire

aucun avantage si ce n'est pour les détails de l'initialisation de la capsule. Nous allons voir à la prochaine section que les machines à états sont surtout utiles pour modéliser les communications entre les différents modèles de calcul à l'intérieur du transacteur de niveau modèle.

L'environnement de vérification est développé à l'aide de Simulink en parallèle avec la spécification exécutable du système en UML. Les deux modèles sont couplées à l'aide d'un transacteur niveau modèle afin de constituer une première version cohérente et exécutable du prototype fonctionnel, qui est indépendante de toute plateforme. On obtient un système où les données sont produites dans Simulink, traitées dans UML et retournées dans Simulink pour fins d'analyse.

3.3 Développement du transacteur entre Simulink et UML

La communication entre l'environnement Simulink et la spécification exécutable en UML constitue un cas d'assemblage hétérogène où une partie du prototype est exécutée dans un simulateur à événements discrets (la partie Simulink), alors qu'une autre partie est exécutée en temps réel sur un processeur (le modèle UML exécutable). Rappelons que l'expression « temps réel » est utilisée ici en opposition à « temps simulé » et ne signifie pas nécessairement que le modèle atteigne une certaine performance en temps réel. Selon le modèle client-serveur point-à-point présenté dans le chapitre 2, le modèle UML agit comme serveur et le modèle Simulink agit comme client. La synchronisation de ces deux modèles est réalisée par un transacteur de niveau modèle, qui prend la forme d'une capsule dans le modèle UML et d'un bloc de type S-Function dans le modèle Simulink. Une S-Function est un bloc spécial inséré dans un modèle Simulink afin d'exécuter du code C++ durant une simulation. Comme pour tous les autres blocs, l'ordonnanceur exécute le bloc S-Function une fois par cycle de simulation. C'est alors qu'une transaction développée sur mesure en code C++ peut se produire entre un point du modèle Simulink et un point du modèle UML (le code source de ce bloc est reproduit à l'annexe II). Ce bloc sera donc utilisé afin de réaliser le côté client du transacteur.

Puisqu'un transacteur est avant tout une interface, il est possible d'établir son contrat d'interface. En réalité, en tant que médiateur entre deux composants hétérogènes, un transacteur possède deux interfaces distinctes, une pour chaque domaine de modélisation. Le rôle du transacteur est de concilier ces interfaces par des mécanismes de synchronisation et par l'échange de messages. Ces messages peuvent être représentés à l'aide d'un diagramme de séquence tel qu'illustré à la Figure 3.7.

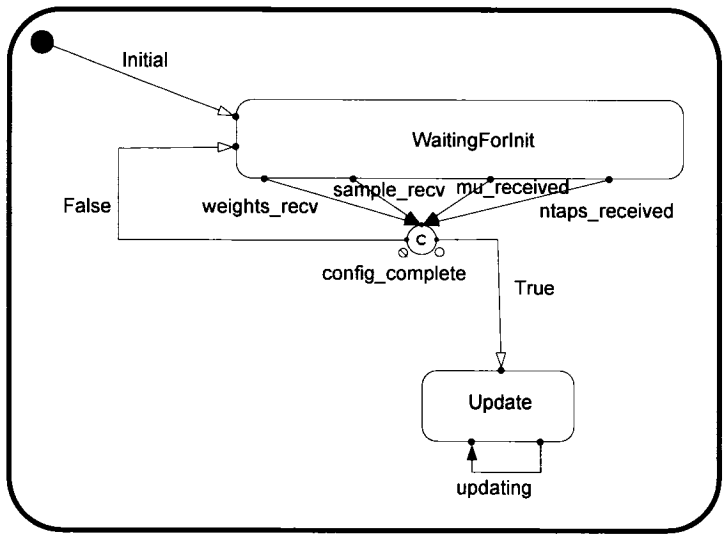

FIG. 3.6: Machine à états de la capsule de mise à jour

Listage 3.1: Code de la transition `updating`

```
1  int i; // Loop iterator
2  ComplexReal error = (ComplexReal) *rtdata;
3  ComplexReal *tempErrorVector;
4
5  tempErrorVector = new ComplexReal[_ntaps];
6
7  //Update weights using complex conjugate of sample value
8  //Substract new value from current value (Check Simulink model)
9
10 for (i=0; i< _ntaps; i++){
11    tempErrorVector[i].re = (_samples[i].re * error.re) - (-1) * (
          _samples[i].im * error.im);
12    tempErrorVector[i].im = (_samples[i].re * error.im) + (-1) * (
          _samples[i].im * error.re);
13 }
14
15 for (i=0; i< _ntaps; i++){
16    _weights[i].re = _weights[i].re - ( tempErrorVector[i].re * _mu);
17    _weights[i].im = _weights[i].im - ( tempErrorVector[i].im * _mu);
18 }
19
20 eqCtrl_comm.updateDone().send();
21
22 delete(tempErrorVector);
```

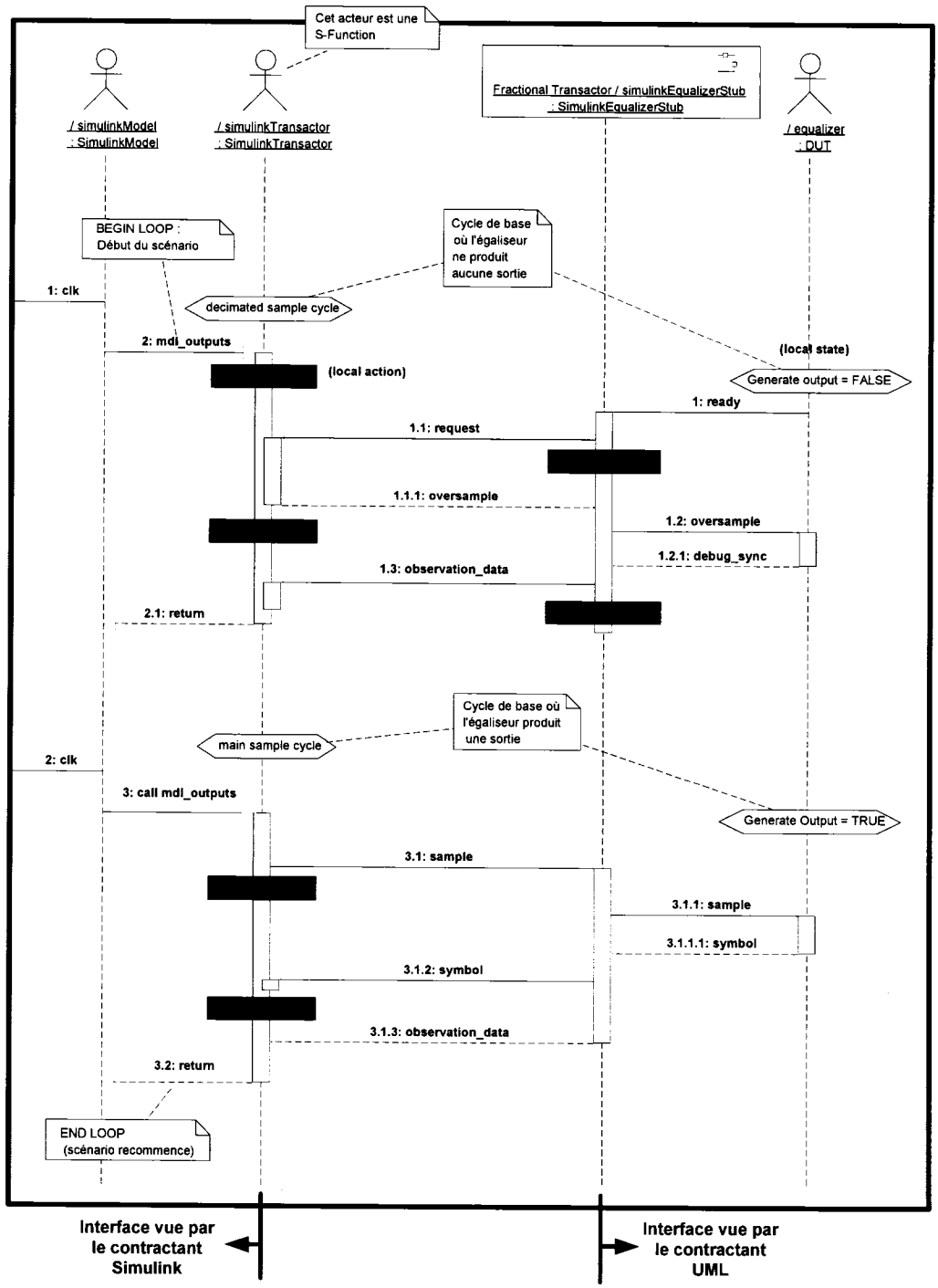

FIG. 3.7: Diagramme de séquence des messages entre le modèle Simulink et le modèle UML

Le diagramme contient quatre acteurs représentés horizontalement, soient les deux interfaces du transacteur ainsi que leur contractant respectif. Chacun des contractants voit une interface différente, selon qu'il se trouve du côté Simulink ou du côté UML. La séquence des messages illustre bien que deux cycles de base sont répétés en boucle : le premier, initié par le coup d'horloge 1 : *clk*, fournit un échantillon fractionnel du flot de données (obtenu par sur-échantillonnage) et le deuxième, initié par le coup d'horloge 2 : *clk*, fournit un échantillon principal. Bien que chaque cycle se termine par le renvoi de données d'observation par le biais des messages 1.3 et 3.1.3, seul le deuxième cycle fournit une décision à la sortie de l'égaliseur par le biais du message 3.1.2. Chaque cycle de base est ainsi encadré par les messages échangés entre le modèle simulé et le modèle exécuté et la synchronisation est assurées par des lectures bloquantes.

Du côté UML, la capsule du transacteur est ajoutée au même niveau hiérarchique que la capsule de l'égaliseur et prend ainsi la forme d'un générateur / analyseur de vecteurs (Figure 3.8). À l'interne, un socket TCP est implanté dans sa propre sous-capsule afin de s'occuper des échanges de paquets avec le socket TCP correspondant de la S-Function. Le fait d'implanter le socket dans sa propre capsule permet de l'isoler dans un processus concurrent, une fonctionnalité qui pourrait être utilisée éventuellement pour effectuer le pipelinage du traitement. Les données reliées à une seule transaction sont sérialisées à l'intérieur d'un même paquet afin d'améliorer la performance des transacteurs. Ceci est facilement réalisé par l'utilisation d'une structure de données commune entre la moitié Simulink et la moitié UML du transacteur. Par exemple, la structure *EqInData* (listing 3.2), échangée via les messages 1.1.1 et 3.1 de la Figure 3.7, contient toutes les données transmises de Simulink à UML lors d'une demande de traitement d'un échantillon.

La capsule utilise également une machine à états afin de synchroniser les messages entre le socket et le contrat d'interface de l'égaliseur UML (Figure 3.9). Une machine

Listage 3.2: Structure d'échange de données

```
1  struct EqInData
2  {
3     ComplexReal in;           // Échantillon à filtrer
4     ComplexReal inTraining;   // Données d'entraînement
5     Boolean hold;             // Geler la mise à jour
6     Boolean reset;            // Remise à zéro des registres
7     Boolean mode;             // Mode entraînement (V) ou poursuite (F)
8     int pad_1;                // Alignement de la structure sur 8 octets
9     double delta;             // Facteur d'ajustement de l'erreur
10 };
```

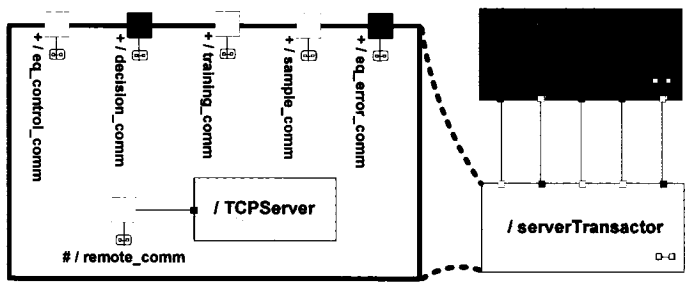

FIG. 3.8: Interconnexion de la capsule transacteur et de la capsule égaliseur

à états ayant le même rôle de synchronisation, mais sans représentation graphique, est également implantée dans la moitié S-Function du transacteur.

Un prototype hétérogène équipé de ce transacteur peut donc être exécuté de façon cohérente et faciliter la validation initiale de la spécification. Cette version du prototype est dite indépendante de toute plateforme (*Platform-Independent Model, PIM*) puisqu'elle ne tient pas compte des contraintes matérielles de la plateforme matérielle cible. Seule la fonctionnalité du système est implantée, d'où l'appellation prototype fonctionnel. Notons enfin que le déterminisme des interactions entre les principaux acteurs du prototype ne justifie pas la définition d'un contrat de synchronisation détaillé.

Un exemple d'utilisation du prototype dans les premières phases du développement a été le tracé de l'erreur quadratique moyenne de l'égaliseur, tel qu'illustré à la Figure 3.10. L'erreur quadratique moyenne est l'indice principal du bon fonctionnement d'un l'égaliseur adaptatif linéaire. Au fil de son opération, l'égaliseur

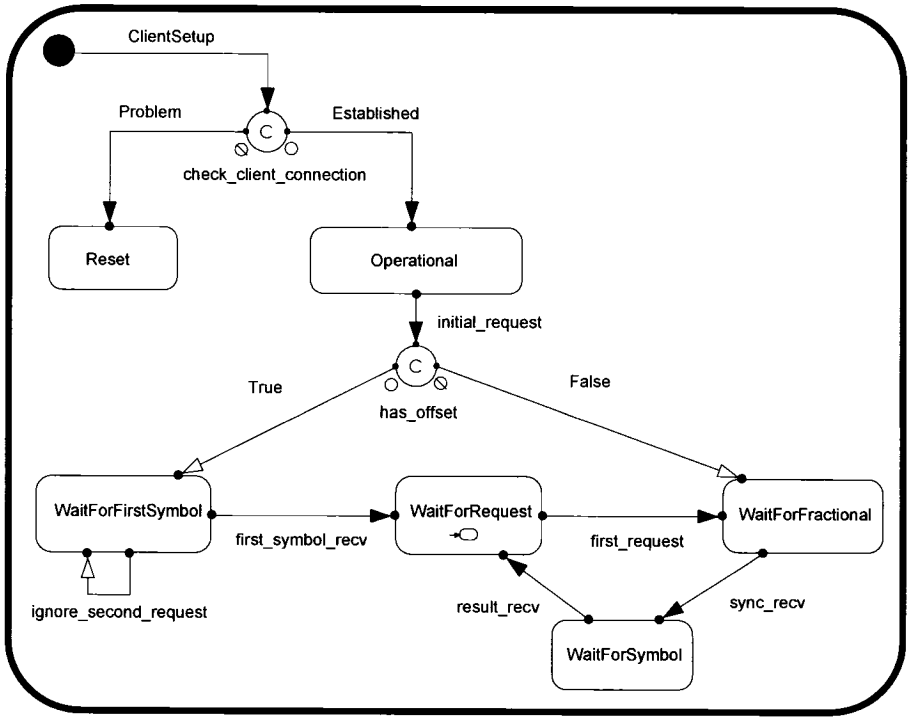

FIG. 3.9: Machine à états de synchronisation des messages du côté UML

réussit, par l'adaptation continue de ses coefficients, à cerner les caractéristiques du canal et à fournir une estimation de plus en plus précise du symbole désiré, réduisant de ce fait l'erreur quadratique moyenne. La cohérence entre l'environnement Simulink et le modèle UML a permis le tracé dynamique de cette courbe à l'aide des blocs *Calcul EQM* et *SCOPE_EQM* de la Figure 3.4, ce qui a grandement facilité le développement et la validation de l'algorithme implanté.

3.4 Modèle PSM et raffinement itératif

Le modèle PIM est appelé à être spécialisé en un modèle PSM par un processus de raffinement itératif. La plateforme matérielle cible utilisée dans le cadre de ce travail est la plateforme *Nios development board, Stratix Professional Edition*, fabriquée

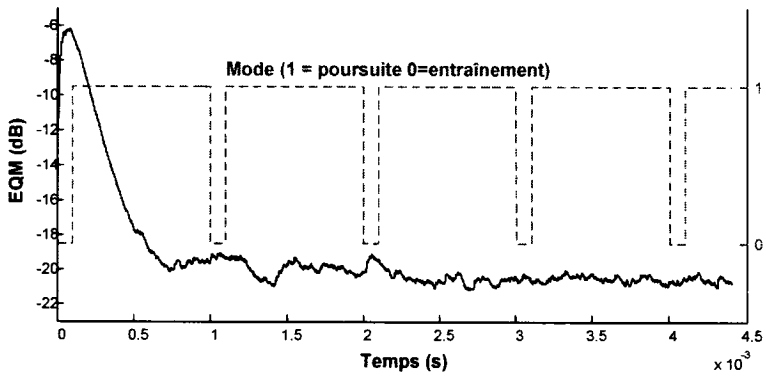

FIG. 3.10: Tracé dynamique de l'erreur quadratique moyenne à l'aide de Simulink

par Altera et adaptée au développment de systèmes embarqués. La plateforme possède les caractéristiques suivantes :

- Un FPGA Stratix EP1S40F780C5 (41250 éléments logiques)
- Un noyau de processeur Nios 32 bit
- 4 kilo-octets d'antémémoire pour les données et 4 kilo-octets pour les instructions
- Une infrastructure de bus Avalon
- 1 mégaoctets de RAM
- 16 mégaoctets de SDRAM
- 8 mégaoctets de mémoire flash
- Une interface MAC/PHY Ethernet et une bibliothèque native au Nios pour son exploitation

Le système en cours de développement, l'égaliseur en l'occurrence, devra éventuellement être intégré sur cette plateforme. Le rôle du raffinement itératif est de déterminer quelles parties du PIM s'exécuteront sur les différentes ressources offertes par la plateforme afin de respecter les contraintes de performance. La définition d'un chemin systématique entre le PIM, le PSM et la plateforme, par le biais d'un modèle UML paramétrable, constitue l'aboutissement de la plateforme de développement rapide proposée dans ce travail. Ce chemin systématique est illustré par les

Figures 3.11, 3.12 et 3.13.

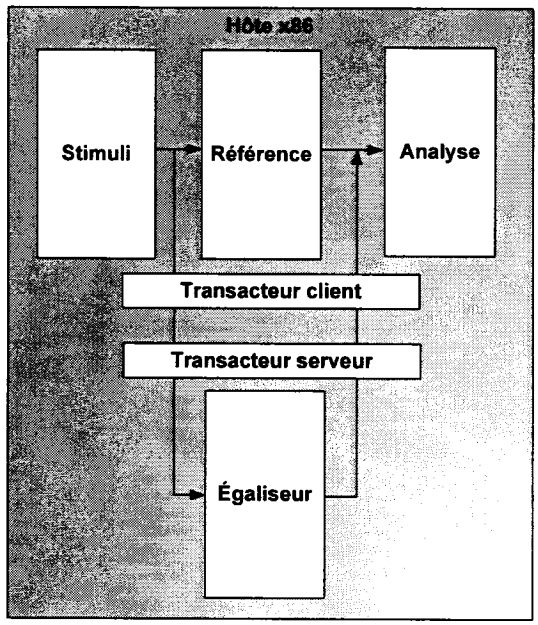

FIG. 3.11: Configuration monoprocesseur

La première version du prototype est entièrement exécutée de façon native sur une machine hôte, tel que l'illustre la Figure 3.11. Cette configuration peut servir à la validation du PIM grâce à une exécution rapide et facilement contrôlable et observable. Le mécanisme de socket est configuré en mode boucle locale à l'aide de l'adresse IP 127.0.0.1. Un modèle de type PSM est ensuite produit pour la configuration de la Figure 3.12. Ce modèle est explicitement ciblé vers le processeur embarqué Nios de la plateforme Altera. La préparation du modèle pour une exécution distribuée requiert les deux étapes suivantes :

1. Recompiler le code généré du UML vers le processeur Nios.
2. Changer l'adresse IP du transacteur client pour celle de la plateforme Altera.

Le dernier raffinement réalisé dans le cadre de ce travail permet d'intégrer dans la boucle de vérification une implantation matérielle sur FPGA de l'égaliseur. Dans cette configuration, illustrée à la Figure 3.13, la synchronisation des opérations de

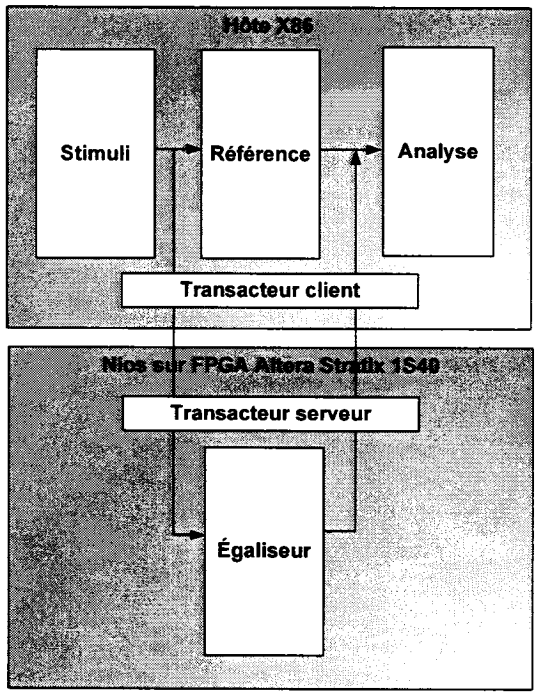

FIG. 3.12: Configuration multiprocesseurs

lecture et d'écriture entre Simulink et l'égaliseur est toujours effectuée par le transacteur à l'aide de lectures bloquantes. Cependant, la capsule UML de l'égaliseur est implantée en tant que pilote logiciel qui effectue la quantification des données provenant de Simulink et les relaye à une implantation matérielle de l'égaliseur. L'interaction entre ce pilote et le composant matériel s'effectue à l'intérieur même du FPGA via l'infrastructure de communication Avalon[1]. Le pilote examine par polling un registre de statut qui indique qu'un symbole filtré est disponible (seulement un cycle sur deux, rappelons-le, selon la configuration en mode fractionnel). Le symbole est lu et retourné à Simulink via le mécanisme de socket habituel pour fins de vérification et d'analyse.

Cette configuration de type matériel-dans-la-boucle est un exemple concret de réutilisation d'un banc d'essai et constitue un pas vers l'implantation finale du système

[1]http://www.altera.com/products/ip/processors/nios/features/nio-avalon_bus.html

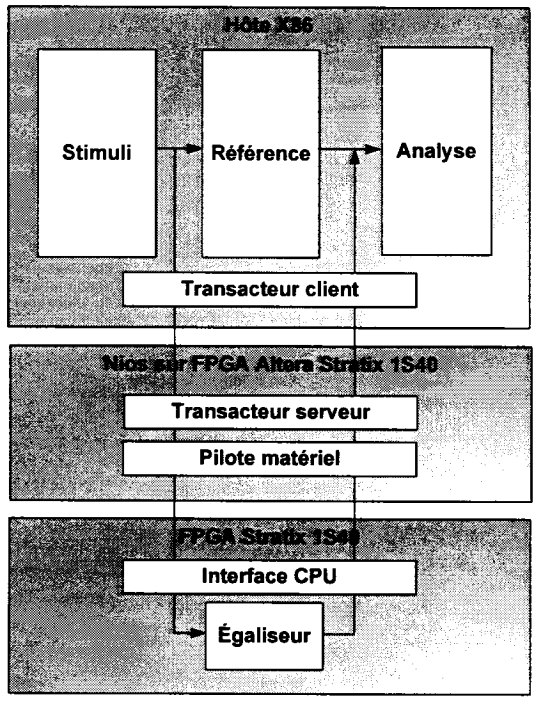

FIG. 3.13: Configuration multiprocesseurs avec co-processeur matériel

sur la plateforme cible. De plus, le pilote matériel, qui offre la même interface que la capsule UML des autres configurations, pourra agir en tant que représentant de l'implantation matérielle pour faciliter la réutilisation de l'égaliseur dans d'autres modèles.

3.5 Exécution, profilage et caractérisation du modèle PSM

Avant de débuter le processus de profilage, les contraintes de performance du système peuvent être ajoutées au contrat d'interface. Ces contraintes sont des paramètres du modèle PIM qui guident l'allocation des ressources de traitement lors du raffinement en PSM. Le contrat d'interface de la capsule égaliseur est donc augmenté des contraintes décrites au tableau 3.2.

Bien que l'aspect temps n'ait pas été introduit explicitement dans les modèles, il

TAB. 3.2: Contrat d'interface au niveau performance

Contrainte	Description
Capacité d'échantillonnage	L'égaliseur doit être en mesure de traiter 8000 échantillons /s.
Débit	L'égaliseur doit fournir un débit binaire de 50 kb/s.
Latence	La latence du système devrait être moins de 300 ms.
Bit Error Rate [a] (BER)	La disponibilité du service radio doit respecter la condition BER < 10e-6

[a]Le BER est donnée à titre d'exemple seulement, il ne sera pas évalué dans le cadre du présent travail.

est possible de vérifier les contraintes de performance par le biais de métriques, qui sont des quantités mesurables qui caractérisent une implantation. Les métriques peuvent être évaluées de façon statique, comme dans le cas de la surface occupée par l'égaliseur, ou de façon dynamique par l'exécution et le profilage du prototype fonctionnel. Le débit binaire, par exemple, est évalué selon les équations suivantes :

$$d = \frac{n_{bits}}{t_{traitement}} \tag{3.2}$$

$$t_{traitement} = \frac{C\langle b\rangle - C\langle a\rangle}{f} \tag{3.3}$$

où n_{bits} représente le nombre de bits produits, $C\langle x\rangle$ représente la valeur du compteur de cycles d'horloge lors de l'événement final et initial de la chaîne de traitement sous étude et f représente la fréquence de l'horloge. Dans le cas d'une implantation logicielle, un compteur de cycles d'horloge CPU peut être interrogé afin de capturer $C\langle a\rangle$, $C\langle b\rangle$ ainsi que des valeurs intermédiaires utiles au profilage. Il suffit pour ce faire d'instrumenter les machines à états de l'égaliseur à l'aide de code inséré dans certaines transitions. La valeur de $t_{traitement}$ peut être estimée en mesurant

le laps de temps minimal entre deux productions de symboles (origine du message 3.1.1.1) de la Figure 3.7 avec comme hypothèse un coût de communication nul, ce qui revient à additionner le temps où l'égaliseur est actif (symbolisé par les deux bandes de focus du contrôle). La période d'échantillonnage moyenne estimée est obtenue en divisant $t_{traitement}$ par deux puisqu'un symbole est produit tous les deux échantillons. Cette période sera inversée afin d'obtenir la capacité d'échantillonnage moyenne estimée de l'égaliseur.

Il est également intéressant de caractériser le coût de communication attribuable au mécanisme de transaction entre les composantes hétérogènes. Le coût de communication associé à un échantillon peut être calculé en faisant une moyenne du laps de temps entre la réception du message 2 et l'envoi du message 2.1 et du laps de temps entre la réception du message 3 et de l'envoi du message 3.2 (toujours en se référant à la Figure 3.7).

Une fois les mécanismes de profilage et de caractérisation en place, le prototype fonctionnel est exécuté. Le modèle UML est d'abord exécuté pour agir en tant que serveur, suivi du modèle Simulink qui agit en tant que client. Les deux moitiés du transacteur sont reliées par une connexion TCP et l'exécution cyclique du modèle Simulink est démarrée. L'engin de simulation déclenche chacun des blocs du modèle tour à tour, ce qui produit les données à traiter attendues du côté UML. Les lectures bloquantes de part et d'autre du transacteur assurent la synchronisation du système et un ordonnancement partiel des événements.

Un premier profilage de l'égaliseur est réalisé selon la configuration native proposée à la Figure 3.11 sur une machine de développement hôte de type Pentium. Ce processeur contient un compteur de cycles intégré dont la valeur peut être capturée à l'aide de l'instruction assembleur RDTSC. Les éléments qui sont profilés sont les deux cycles de bases présentés à la Figure 3.7, le temps total de traitement tel que

TAB. 3.3: Sommaire des résultats de profilage selon les trois configurations proposées

	Config. 1 (fig. 3.11)	Config. 2 (fig. 3.12)	Config. 3 (fig. 3.13)
Type de configuration	PIM (native)	PSM	PSM
Matériel	Pentium 2.80 GHz	Nios 50 MHz	Nios et LTE matériel 50 MHz
Capacité d'échantillonnage (échantillons par seconde)	26 200	112	8 333 000
Débit binaire (QPSK : 4 bits par symbole)	52,5 kb/s	0,225 kp/s	16,67 Mb/s
Latence (ms)	0,38	88,94	0,001
Débit global (symboles par seconde)	91	18,5	20,8

vu par Simulink pour chacun de ces cycles, ainsi que le temps global requis pour le traitement de 2000 échantillons.

Il est possible de profiler le prototype selon les configurations distribuées présentées aux Figures 3.12 et 3.13 selon les mêmes métriques. Un compteur implanté en matériel, qui partage la même horloge que le Nios, est utilisé pour profiler l'implantation sur le processeur. Quant à l'implantation matérielle, le nombre de cycles est connu d'avance, seuls restent les coûts de communication à évaluer. Les résultats du profilage de l'égaliseur sont résumés dans le tableau 3.3.

Selon les données de cette table, les configurations 1 et 3 sont en mesure de respecter le contrat d'interface de l'égaliseur. Avec un débit de 52,5 kbps, l'égaliseur exécuté sur le processeur Pentium est capable de filtrer des échantillons audionumériques en temps réel. Quant à la configuration 2, qui effectue l'égalisation sur le processeur Nios cadencé à 50 MHz, elle ne peut fournir ni le débit ni le taux d'échantillonnage prescrit par le contrat.

Ces résultats ne tiennent pas compte du coût de communication inféré par l'utilisation des transacteurs. Ainsi, selon la configuration 1, le coût de communication est d'environ 0,44 ms, soit 10 fois le temps de traitement d'un symbole. Pour la configuration 2, le coût de communication explose à environ 28,4 ms, une augmentation due à la connexion physique par lien Ethernet 100 Mb/s entre l'ordinateur hôte qui exécute Simulink et le processeur Nios de la plateforme. Dans le cadre de la configuration 3.13, l'égaliseur est implanté sous forme matérielle dans le FPGA. La production d'un symbole prend exactement 12 cycles de l'horloge réglée à 50MHz, ce qui fournit une performance maximale théorique de 8,33 méga-échantillons par seconde. Cependant, le coût de communication du transacteur comprend non seulement le temps de transfert via le lien Ethernet (inclus dans l'overhead Simulink), mais aussi le temps d'exécution des différentes fonctions du pilote de l'égaliseur et le temps de transfert par le bus Avalon (inclus dans l'overhead UML). Tout ceci représente un coût de communication d'environ 40 ms pour la production d'un symbole, ce qui réduit considérablement le gain de performance apporté par l'implantation matérielle de l'égaliseur. Ces résultats sont illustrés à la Figure 3.14.

Il est intéressant de remarquer, à la Figure 3.14, que le surcoût associé à Simulink est relativement plus important pour la configuration 3 que pour la configuration 2, alors que rien n'a changé dans le mécanisme de communication. Ceci est probablement dû à un délai incompressible de communication et de synchronisation entre Simulink et la plate-forme Nios. Enfin, le débit global moyen a été mesuré pour le traitement de 2000 échantillons et les résultats sont indiqués à la dernière rangée de la table 3.3. Le débit varie largement selon les conditions d'expérimentation et il est influencé par une multitude de facteurs : la longueur du câble Ethernet, le nombre de processus concurrents qui s'exécutent sur le système Windows (sur lesquels nous n'avons qu'un contrôle limité), l'état de l'antémémoire, le nombre de fenêtres ouvertes, la technique de profilage utilisée, etc. L'attribution de différentes

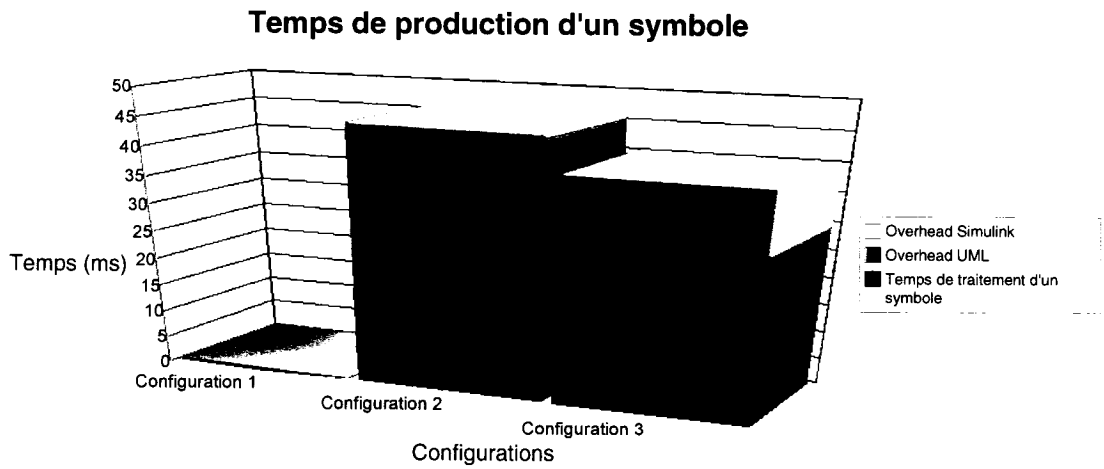

Tableau des résultats (temps en ms)

	Configuration 1	Configuration 2	Configuration 3
Temps de traitement vu par Simulink	0,523	46,031	40,049
Temps de traitement d'un symbole	0,046	17,597	0,000
Temps de traitement vu par UML	0,084	45,207	29,155
Overhead UML	0,038	27,609	29,155
Overhead Simulink	0,438	0,824	10,893

FIG. 3.14: Résultats du profilage selon les trois configurations

priorités aux processus Windows permettrait de stabiliser le débit et d'améliorer les caractéristiques temps réel du prototype. Cet aspect n'a pas été exploité et la valeur du débit global est fournie à titre indicatif seulement.

Comme il fallait s'y attendre, les communications constituent le principal goulot d'étranglement lors de l'exécution du prototype. Le surcoût de communication entre l'environnement de vérification et l'égaliseur pourrait être diminué de façon significative en regroupant plusieurs cycles de base pour permettre le transfert de leurs données en rafale. Ce grossissement de la granularité du traitement aurait le contre-effet de limiter la résolution de la rétroaction entre l'égaliseur et un système en amont. La performance du prototype pourrait également être améliorée en faisant appel à un mécanisme d'interruption plutôt qu'à des lectures bloquantes, afin de synchroniser les différents traitements du système. Sur la plateforme Nios, une interruption matérielle peut être activée lors de la réception d'un message par l'interface réseau. Une machine hôte équipée d'un système d'exploitation utilisera plutôt une interruption logicielle, notamment via le signal SIGIO. Un mécanisme

d'interruption aurait comme principal avantage de libérer des ressources pour les processus qui sont concurrents à la lecture bloquante. Quant au traitement des échantillons, une accélération serait également possible par l'utilisation d'un processeur spécialisé dans le traitement DSP plutôt qu'un processeur général comme le Nios.

Les résultats présentés dans ce chapitre ont démontré que la définition et l'implantation d'interfaces en UML peut faciliter la réutilisation, permettre le raffinement itératif et permettre la validation et le profilage. Plusieurs autres observations et recommandations peuvent être faites à partir de ces résultats, ce qui constitue l'objet du prochain et dernier chapitre.

CHAPITRE 4

ANALYSE DES RÉSULTATS ET DISCUSSION

Le chapitre précédent a démontré qu'il était possible de découpler la spécification d'interface d'un composant de son implantation à l'aide d'une capsule UML et de mécanismes de transaction multi-niveaux. Le composant d'égalisation a été conçu dans le respect du contrat d'interface de la capsule, ce qui permet de l'importer dans toute autre spécification où le même contrat pourrait être utilisé. Une bibliothèque de composants conçus selon la même méthodologie constitue une plateforme de conception qui permet l'assemblage rapide de blocs pré-caractérisés à plusieurs niveaux d'abstraction (du niveau comportemental jusqu'à l'implantation).

L'application de cette méthodologie à la conception et l'implantation d'un composant réel a permis d'observer certains éléments, avantages ou inconvénients, qui sont l'objet de l'analyse de ce chapitre. Les limites connues des techniques développées et des suggestions pour l'amélioration de la méthodologie concluent le chapitre.

4.1 Production vs. réutilisation de blocs IP

L'utilisation de la méthodologie a démontré la spécification de l'interface d'un composant d'égalisation puis son implantation, son raffinement et sa caractérisation en fonction d'une plateforme cible. Cette préparation descendante du bloc, d'un niveau d'abstraction supérieur à un niveau inférieur, en vue d'une réutilisation ultérieure, ne représente que l'aspect *production* d'un bloc IP. Un aspect complémentaire est la réutilisation telle quelle de blocs dans un design différent. Nous allons présenter

les éléments qui font qu'un bloc préparé selon la méthodologie est plus facilement réutilisable dans un autre design.

Le raffinement itératif du modèle PIM en PSM constitue un cheminement descendant qui est naturel lorsqu'un design complexe est abordé. Très tôt cependant, des caractéristiques provenant des niveaux d'abstraction inférieurs peuvent être insérées dans les modèles afin de guider l'implantation de certains composants. Par exemple, dans le cas d'un récepteur RRL, une interface de communication compatible avec la norme IEEE 802.11 peut être achetée d'un tiers parti. Ceci constitue l'essence même d'une réutilisation efficace à la base de la productivité du design. La description et la matérialisation du contrat d'interface d'un IP réutilisable, tel que cette interface 802.11, sous la forme d'une capsule UML permet son intégration dans le prototype fonctionnel à un haut niveau d'abstraction avec l'assurance qu'un chemin systématique vers l'implantation de ce bloc existe. Quant à cette implantation pré-caractérisée et pré-vérifiée du bloc IP, elle peut-être intégrée dans le prototype fonctionnel de façon transparente à l'aide d'un transacteur de niveau modèle et ainsi contribuer à l'exécution du prototype.

La convergence des approches ascendante et descendante en un point médiant entre la spécification et l'implantation constitue l'approche "meet-in-the-middle" prescrite par la conception basée plateforme. L'atteinte de ce point est facilitée par l'utilisation d'interfaces qui font l'encapsulation des détails d'implantation. Un prototype fonctionnel décrit à haut niveau d'abstraction peut donc être contraint de façon itérative afin de satisfaire aux spécifications de performance par l'ajout de composants implantés et encapsulés. Ainsi, le composant d'égalisation qui a été développé pourra être réutilisé dans la conception d'un récepteur RRL soit au niveau fonctionnel, tel que le permet son implantation en C++ sur Pentium, soit lors de l'implantation sur un processeur embarqué ou un FPGA, et ceci à partir d'une spécification unique et exécutable en UML.

4.2 Niveau d'abstraction et aspect temps

La construction du modèle PIM débute habituellement à un niveau élevé d'abstraction où la signalisation détaillée entre composants n'est pas prise en compte. Les communications sont effectuées à un niveau transactionnel par l'échange de messages sur les ports de capsules. Ce niveau est approprié à l'exploration algorithmique et permet la construction d'un modèle compact et rapide mais sans aucune information d'implantation, donc peu précis.

Cependant, la flexibilité de la communication par message permet de descendre aussi bas que nécessaire dans les niveaux d'abstraction, tel que présenté par (Cai et Gajski, 2003). Ainsi, un bus complet peut être modélisé sous forme de capsules UML et servir d'infrastructure de communication. À un haut niveau d'abstraction, ce bus est caractérisé par une approximation du temps, une plage d'adresse logique ainsi qu'un mécanisme d'arbitrage. En tenant compte des caractéristiques temporelles des protocoles utilisés, une modélisation précise de la contention et de la performance des communications est possible. Une telle modélisation peut s'effectuer sur la base d'une analyse statique du modèle ou sur la base du profilage du prototype exécuté sur une plateforme réelle. Le concepteur doit choisir l'approche privilégiée. Le raffinement des caractéristiques temporelles peut être poursuivi jusqu'à l'obtention d'un modèle de type RTL précis au niveau temps et cycle.

Un raffinement inégal des capsules d'un modèle provoque l'apparition de différents domaines d'horloge. Par exemple, considérons le cas où une chaîne de traitement contient deux traitements indépendants a et b et un troisième traitement c qui dépend de a et de b. Un composant précis au niveau cycle prend C_a cycles pour compléter le traitement a, alors qu'un autre, moins précis, prend $\pm C_b$ cycles pour compléter le traitement b. Si les traitements a et b sont concurrents, la relation entre les horloges doit être clairement établie, ou un mécanisme de synchronisation

par sémaphores doit être utilisé, afin que le traitement c sache à quel moment ses entrées a et b sont valides. De même, si le transfert des données produites par a et b et consommées par c est effectué via un modèle de bus avec contention, toutes les communications provenant des autres traitements et qui transitent par le même bus doivent être caractérisées. Pour un système hétérogène sur puce, cela constitue un véritable défi qui ne peut être résolu que par l'adoption d'une méthode systématique pour l'implantation des communications.

Dans le cas où la synchronisation du système repose sur des événements séquentiels et ordonnés de façon causale, comme le veut l'hypothèse simplificatrice utilisée dans ce travail, différents domaines d'horloge peuvent quand même co-exister. Par exemple, selon une des configurations présentées, les données de test proviennent de l'environnement Simulink, exécuté sur le processeur Pentium cadencé à 2.80 GHz, alors que le traitement des données est effectué par le processeur Nios cadencé à 50 MHz. Tel que mentionné dans le paragraphe précédent, l'introduction de la parallélisation des traitements et de l'aspect temps dans le système exigera la résolution de tous les temps de traitement et l'utilisation de mécanismes de synchronisation plus fins. Le temps devra être connu (normalement, ceci est le cas pour un traitement réalisé en matériel) ou mesurable / profilable (dans le cas d'un traitement logiciel) pour chaque traitement du système et ce selon une mesure commune et correspondant à la résolution la plus fine disponible. Un profil détaillé des temps de traitement pourra être établi afin de permettre la planification des tâches ainsi que l'estimation et l'optimisation de la performance du système. Si ce profil doit être déterminé par modélisation, le modèle peut devoir être uniformisé à la résolution du composant le plus précis. Cependant, une caractérisation de l'exécution du modèle peut fournir des profils de temps précis sans homogénéisation et fait levier sur le prototype fonctionnel.

4.3 Lacunes et améliorations possibles

Cette section décrit certains aspects des techniques de conception présentées dans ce travail qui pourraient faire l'objet d'une amélioration ou d'une étude plus approfondie.

4.3.1 Multiplicité des composantes hétérogènes

Le développement du prototype fonctionnel d'égaliseur linéaire s'est fait sur la base de trois composants hétérogènes : l'environnement de test Simulink, le modèle logiciel exécutable de l'égaliseur et l'implantation matérielle de l'égaliseur. L'utilisation de *socket* pour l'implantation des transacteurs a permis l'exécution du prototype selon une configuration mono-processeur et ensuite selon une configuration distribuée. Le fait que protocole TCP/IP soit utilisé permet la multiplication des transacteurs à travers le prototype. En effet, un transacteur constitue un lien de communication point à point défini par une adresse IP et un port TCP unique. Un numéro de port TCP est décrit sur 16 bits, ce qui offre 65 535 ports différents pour une même adresse IP (dont certains sont réservés pour des applications courantes et ne devraient pas être utilisés).

Bien que le mécanisme des transacteurs offre une telle abondance de connexions, cet aspect n'a pas été mis à profit dans le prototype. Par exemple, un deuxième transacteur aurait pu être introduit entre le modèle UML et un outil externe de statistique afin d'exporter et d'analyser les coefficients de l'égaliseur à chaque cycle de traitement. Cependant, la simplicité du système sous développement ne justifiait pas l'utilisation de ce deuxième pont de communication. La multiplicité des transacteurs requiert également une gestion plus évoluée des sockets, impliquant le multi-threading et la gestion d'interruptions. Une méthode systématique pour la

distribution et l'interconnexion des composants doit également être mise en place. Cette problématique est illustrée par un exemple à la Figure 4.1. Le déploiement de gauche nécessite l'introduction d'un transacteur de niveau modèle pour réaliser la communication entre les capsules A et B, exécutées dans deux applications différentes. Le déploiement raffiné de droite requiert également ce transacteur, mais cette fois un deuxième processeur est dédié à l'exécution des capsules C et D. Une question intéressante est de déterminer l'emplacement préférable du transacteur qui reliera ces capsules au reste du prototype. Deux options sont possibles : puisque les capsules C et D sont exécutées dans le même contexte, aucun transacteur n'est requis *entre elles*, ce qui permet de les grouper en une seule entité et d'utiliser un seul transacteur entre la capsule B et cette nouvelle entité C+D ; d'autre part, il est possible de dédier un transacteur à chacune des capsules C et D, ce qui a comme avantage de conserver l'intégrité de la structure globale UML mais à un coût de communication plus élevé. Chacune des options devrait être possible et le choix devrait être fait en fonction des critères d'observabilité et de performance désirés.

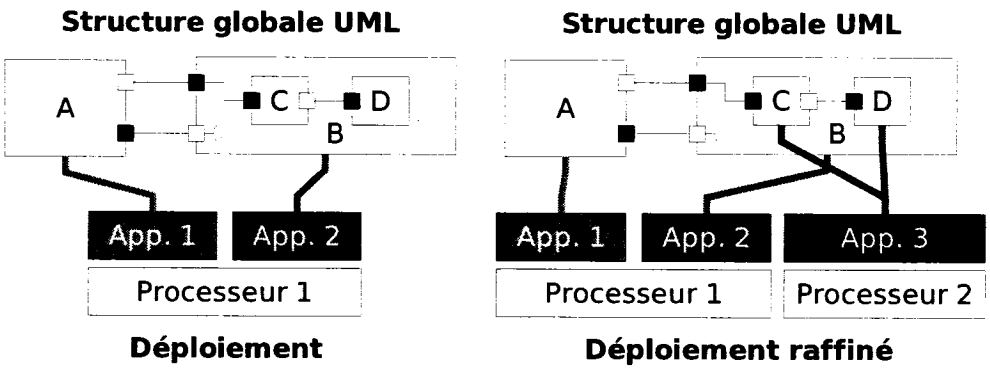

FIG. 4.1: Différents déploiements d'un même modèle

4.3.2 Pipelinage et parallélisation du traitement

Les événements qui ont lieu dans le prototype fonctionnel sont tous linéarisés afin de faciliter en premier lieu la maîtrise de l'algorithme. Cette simplification engendre

un gaspillage des ressources de traitement, qui sont souvent retenues par les lectures bloquantes de synchronisation. Une analyse des dépendances de données propre à l'application révélerait qu'une certaine partie du traitement peut être pipelinée, améliorant sensiblement la performance du système.

La parallélisation repose sur la disponibilité de plusieurs unités de traitement au même moment. Ceci correspond à la configuration multiprocesseurs du protoype fonctionnel ou, plus généralement, à la multiplication des unités de traitement matériel. Cependant, le UML étant à priori un langage à haut niveau d'abstaction, son utilisation influence la résolution à laquelle peut être appliquée le parallélisme ou le pipelinage des traitements. Ainsi, il est plus aisé de pipeliner des traitements de résolution grossière que des traitements fins. Dans le cas de l'égaliseur linéaire en mode fractionnel par exemple, le traitement d'un échantillon sur deux ne produit pas de résultat à la sortie (voir le diagramme de séquence de la Figure 3.7). Il est donc possible de pipeliner le traitement de cet échantillon (traitement 1) avec la production de l'échantillon suivant dans l'environnement Simulink (production 2). Ce pipelinage élémentaire de processus à résolution grossière est illustré à la Figure 4.2.

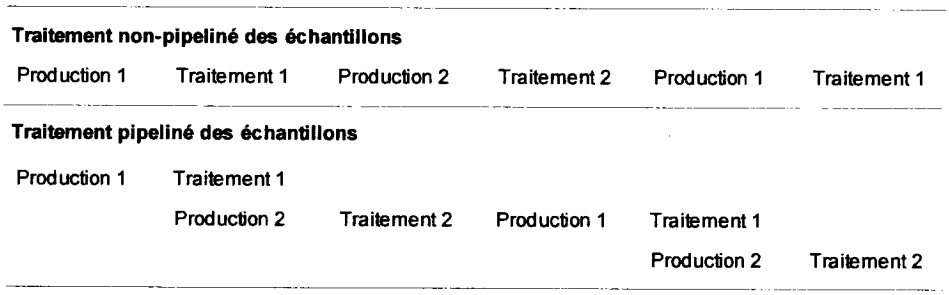

FIG. 4.2: Pipelinage élémentaire de la production et du traitement des échantillons

En se référant à la Figure 3.7, on voit que le pipelinage est réalisé en émettant le message 1.3 avant le message 1.2. À l'opposé, il est difficile de modéliser la parallélisation de la multiplication associée à la boucle de filtrage, par exemple, qui

constitue un traitement à résolution plus fine. Cette tâche peut être accomplie par un compilateur spécialisé ou par une instruction optimisée du processeur, tel qu'une TIE (Tensilica Instruction Extension) dans le cas du processeur Xtensa (Gonzalez, 2000).

Dans le cas d'une parallélisation du processus, la configuration d'un transacteur en mode point-multipoint (multicast ou broadcast) pourrait permettre une communication efficace entre plusieurs unités de traitement ou processeurs. Par exemple, les données provenant de l'environnement de vérification pourraient être envoyées à une banque d'égaliseurs spécialisés par le biais d'un transacteur en mode multicast, réalisant un système de multi-égalisation (Dumais et al., 2004). Dans le cas où plusieurs processus concurrents interagissent avec une même interface, il peut être nécessaire d'établir un contrat au niveau synchronisation.

4.3.3 Raffinement et implantation des transacteurs

Les transacteurs niveau modèle font partie intégrante du processus de raffinement. Ainsi, un transacteur qui réalise la liaison entre deux composants hétérogènes peut lui-même être affecté à une ressource de communication de la plateforme cible. Par exemple, la moitié client d'un transacteur peut être raffinée en un maître de bus AMBA-AHB alors que la partie serveur devient une interface esclave. Le support arrière du transacteur est alors intégré au bus AMBA lui-même, permettant la transmission des messages selon un algorithme d'arbitrage particulier. Dans le cas d'un transacteur qui agit comme interface entre le logiciel et le matériel, il s'agit alors d'intégrer les routines du support arrière client dans un pilote ou une routine d'interruption. Le raffinement des transacteurs ne devrait pas modifier les clauses établies du contrat d'interface. Le raffinement est également possible entre un transacteur de niveau modèle et les ressources d'un réseau sur puce (NoC). L'ensemble

des transacteurs présents dans un prototype peut donc être consolidé en un médium cohérent de communication sur puce. Cette avenue de recherche prometteuse est, une fois de plus, liée à la résolution des communications dans les systèmes sur puce.

En général, il faut éviter de modéliser explicitement l'infrastructure de communication utilisée dans une puce à l'aide de capsules puisque *1)* cette infrastructure fait partie des ressources (plus adéquatement modélisées par un diagramme de déploiement) et *2)* elle est particulière à une plateforme. La généralité d'un modèle facilite sa réutilisation pour d'autres plates-formes, qu'elles soient basées sur la même infrastructure ou non, ce qui est un des buts poursuivis par la méthodologie.

4.4 Généralisation de la méthodologie à d'autres applications

La méthodologie de développement peut être appliquée telle quelle à tout système dominé par un flot de données régulier. Cette dernière condition découle de la synchronisation par lectures bloquantes implantée dans les transacteurs, qui requiert une certaine régularité dans le traitement. Le développement d'un système dominé par le contrôle exigerait une adaptation du mécanisme des transacteurs qui ne sera pas couverte dans ce travail.

Un exemple de système auquel la méthodologie peut s'appliquer est un processeur pour le traitement et la transmission de vidéo. La tâche du système est de combiner le traitement et la transmission de vidéo numérique selon différent protocoles réseau sur un seul système sur puce. Une implantation possible de ce système est d'utiliser un bus de données à haut débit et un bus de contrôle afin d'échanger des données avec une mémoire partagée et un contrôleur. Une modélisation sommaire des principaux blocs de traitement du système à l'aide de capsules est présentée à

la Figure 4.3.

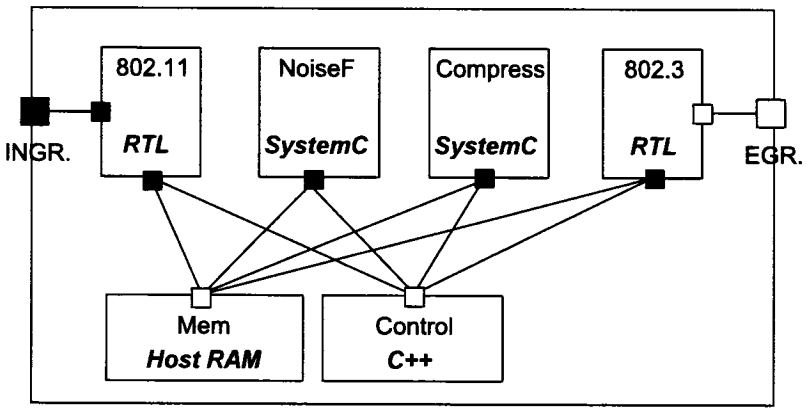

FIG. 4.3: Diagramme de structure d'un processeur vidéo

Dans cet exemple, le flot vidéo provient d'une source externe via une interface de bus Firewire (IEEE1394). Un composant soft-IP pré-conçu qui réalise cette interface est sélectionné et abstrait sous forme d'une capsule de type transacteur niveau modèle afin d'être projeté au niveau d'abstraction UML. L'encapsulation de plusieurs blocs IP de cette façon forme la base d'une plateforme de conception. Dans le cas du processeur vidéo, des composants de filtrage, de réduction du bruit, de compression et de recodage, développés à l'aide de différents outils, sont tous modélisés à l'aide du dénominateur commun que sont les interfaces UML. Lorsque connectés ensemble, ces composants forment une instance de plateforme, telle que celle illustrée à la Figure 4.3.

4.5 UML et la conception de systèmes sur puce

Il convient de terminer cette analyse par des commentaires généraux sur l'utilisation d'UML dans la conception de systèmes sur puce. Le langage offre la versatilité nécessaire pour modéliser des comportements et des structures propres aux systèmes sur puce à différents niveaux d'abstraction. Cependant, son manque de formalisme

et sa structure lâche introduisent trop d'inconnues pour rendre la synthèse matérielle possible. Bien que certaines extensions du langage permettent d'introduire des données de *temps* dans les modèles, la résolution de ces données reste, pour le moment, un processus manuel qu'il est impossible de mener à bien dans le contexte des systèmes sur puce.

Il faut donc combiner UML à d'autres langages afin de s'approcher d'une implantation matérielle, comme fait état la présente recherche. C'est en effet dans un rôle d'organisation du design à un haut niveau d'abstraction que les modèles UML sont le plus utile. Le développement d'outils spécialisés dans l'interprétation de modèles UML pour le déploiement de systèmes sur puce viendra éventuellement compléter ce rôle et étendre les capacités du langage.

CONCLUSION

La mise en oeuvre des techniques pour la conception et la vérification d'un système réel a permis de valider à petite échelle l'approche par interface à la conception de systèmes sur puces. Les spécifications textuelles d'un égaliseur pour radio réalisée par logiciel et de son environnement opérationnel ont été capturées dans un modèle UML à un haut niveau d'abstraction. Les capsules de l'environnement opérationnel ont été exportées dans l'outil Simulink afin de prendre avantage de sa bibliothèque de blocs préconçus pour la génération et l'analyse de signaux de radio numérique. L'exportation s'est faite automatiquement à l'aide d'un script qui transforme les capsules et ports en sous-systèmes et signaux. Le modèle Simulink ainsi créé a été raffiné manuellement par un ingénieur d'application pour servir d'environnement de vérification réutilisable tout au long du développement. Quant aux capsules constituant l'égaliseur lui-même, leur raffinement s'est poursuivi sous forme de diagrammes d'état UML et de code C++ en accord avec les contrats d'interfaces. Cette implantation du système a été vérifiée de façon fonctionnelle à l'aide de l'environnement Simulink, puis des métriques de performances ont été évaluées en fonction d'une plateforme d'implantation cible afin de valider le respect des contraintes de performance.

La performance en temps réel de la spécification exécutable de l'égaliseur a été évaluée à environ 52,5 kbps sur un processeur de type Pentium, soit un débit permettant le traitement d'un signal audio numérique. Le prototype a ensuite été exécuté selon une configuration distribuée, où l'environnement de vérification fut exécuté sur un ordinateur hôte alors que le système sous développement fut exécuté sur un processeur embarqué Nios. Cette configuration de type matériel-dans-la-boucle a permis de démontrer la réutilisation de l'environnement de vérification tout en fournissant un débit en temps réel d'environ 21 symboles QPSK par seconde.

Enfin, le raffinement a été poussé jusqu'à une implantation matérielle de l'égaliseur, qui a été vérifiée à l'aide de la même infrastructure de communication.

La principale contribution de ce travail est d'avoir positionné le langage de modélisation UML dans un flot de conception de système sur puce. Il a ainsi été observé que l'utilisation du langage aide à la spécification structurelle du système et à l'orthogonalisation des aspects fonctions et communication. La définition de contrats d'interface à l'aide de protocoles et de diagrammes a servi à l'élaboration et au raffinement des interconnexions du système. Enfin, les modèles construits en UML sont auto-documentés, expressifs et fournissent un moyen de communication que plusieurs intervenants peuvent utiliser. Par contre, les modèles ne sont pas appropriés pour une représentation précise du temps physique, un aspect essentiel à la synchronisation de circuits à haute vitesse. La dépendance envers un outil de génération de code à partir d'un modèle de même que l'absence de règles de cohérence inter-diagrammes peuvent également être vues comme des faiblesses du langage.

Plusieurs avenues de recherche s'ouvrent à la suite de ce travail. Il serait intéressant, dans un premier temps, de formaliser la relation entre le processus de raffinement et la validité des contrats. En effet, le raffinement peut introduire une variation naturelle dans les caractéristiques d'un composant qui peut invalider un contrat d'interface ou diverger par rapport à un modèle logiciel. Il serait également intéressant d'explorer les capacités du langage UML dans la modélisation et l'attribution des ressources matérielles et d'évaluer l'impact de la parallélisation du traitement sur la spécification. Enfin, l'introduction de l'aspect temps dans les modèles pourrait être facilitée par une recherche sur les mécanismes de résolution des délais de composantes et d'interconnexions à partir de diagrammes UML. Ainsi, le langage constitue un outil intéressant pour une variété de travaux de recherche sur les techniques et méthodes de conception de niveau système.

RÉFÉRENCES

ACCELCHIP (2005). *DSP synthesis.* [Logiciel]. Milpitas, Calif.

ACCELLERA (2004). *SystemVerilog 3.1a Language Reference Manual - Accellera's Extension to Verilog.* Accellera Organization, Napa, CA.

ALQUIER, C., GUERINEAU, S., RIZZATTI, L., BURGUN, L. (2003). « Co-simulation between SystemC and new generation emulator ». *Proceedings of DesignCon.* Santa Clara, Calif.

ANDERSON, T. L. (2004). « Spec raises bar for IP and SoC verification ». *EETimes.* [En ligne]. http://www.us.design-reuse.com/news/news7435.html (Page consultée le 8 juin 2005)

ASHENDEN, P. J. (2002). *The Designer's Guide to VHDL.* 2nd ed.

BALARIN, F., GIUSTO, P., JURECSKA, A., PASSERONE, C., SENTOVICH, E., TABBARA, B., CHIODO, M., HSIEH, H., LAVAGNO, L., SANGIOVANNI-VINCENTELLI, A., SUZUKI, K. (1997). *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach.*

BENINI, L., DE MICHELI, G. (2004). « Networks on chip: a new paradigm for component-based MPSoC design ». *Multiprocessors Systems on Chips.* Morgan Kaufman. 49–80.

BENVENISTE, A., BERRY, G. (1991). « The synchronous approach to reactive and real-time systems ». *Proceedings of the IEEE.* 79, 1270–1282.

BERGERON, J. (2003). *Writing testbenches: functional verification of HDL models.* 2nd ed. Norwell, MA : Kluwer Academic Publishers.

BEUGNARD, A., JÉZÉQUEL, J.-M., PLOUZEAU, N., WATKINS, D. (1999). « Making components contract aware ». *IEEE Software.* 38–45.

BOOCH, G., RUMBAUGH, J., JACOBSON, I. (1998). *Unified Modeling Language User Guide.*

BOULET, P., DEKEYSER, J.-L., DUMOULIN, C., MARQUET, P. (2003). « MDA for SoC embedded systems design, intensive signal processing experiment ». *White Paper.*

BRINGMANN, O., ROSENSTIEL, W., MUTH, A., FÄRBER, G., SLOMKA, F., HOFMANN, R. (1999). « Mixed abstraction level hardware synthesis from SDL for rapid prototyping. ». *Proceedings of the IEEE International Workshop on Rapid System Prototyping.* 114–119.

BUCK, J., HA, S., LEE, E. A., MESSERSCHMITT, D. G. (1992). « Ptolemy: A framework for simulating and prototyping heterogenous systems ». *Int. Journal in Computer Simulation.* 4:2.

CADENCE (2005). *Incisive functional verification platform.* [Logiciel]. San Jose, Calif.

CAI, L., GAJSKI, D. (2003). « Transaction level modeling : an overview ». *Proc. of CODES+ISSS'03.* ACM. 19–24.

CELOXICA (2005). *DK design suite.* [Logiciel]. Abingdon, UK.

CESARIO, W., BAGHDADI, A., GAUTHIER, L., LYONNARD, D., NICOLESCU, G., PAVIOT, Y., YOO, S., JERRAYA, A. A., DIAZ-NAVA, M. (2002). « Component-based design approach for multicore socs ». *DAC '02: Proceedings of the 39th conference on Design automation.* New York, NY, ACM Press. 789–794.

CHANG, H., COOKE, L., HUNT, M., MARTIN, G., MCNELLY, A. J., TODD, L. (1999). *Surviving the SoC Revolution - A Guide to Platform-Based Design.*

CHEN, R., SGROI, M., LAVAGNO, L., MARTIN, G., SANGIOVANNI-VINCENTELLI, A., RABAEY, J. (2003). « 5. UML and platform-based

design ». *UML for Real: Design of Embedded Real-Time Systems.* KAP. 107–126.

CHEVALIER, J., BENNY, O., RONDONNEAU, M., BOIS, G., ABOULHAMID, E., BOYER, F.-R. (2003). « SPACE: A hardware/software SystemC modeling platform including an RTOS ». *Proc. Forum on Design Languages (FDL03).* Frankfurt. 704–715.

CINDERELLA (2005). *Cinderella SDL 1.3.* [Logiciel]. Ballerup, Denmark.

COHEN, B. (2000). *Component Design by Example: a Step-by-Step Process Using VHDL with UART as Vehicle.*

COHN, J. M. (2003). « 11. technology challenges for soc design, an ibm perspective ». *Winning the SoC Revolution, Experiences in Real Design.* KAP. 255–296.

COWARE (2005). *Signal processing workstation (SPW).* [Logiciel]. San Jose, Calif.

CYR, G., BOIS, G., ABOULHAMID, M. (2004). « Generation of processor interface for soc using standard communication protocol ». *Computers and Digital Techniques, IEE Proceedings.* 151:5. 367–376.

DALLY, W. J., TOWLES, B. (2001). « Route packets, not wires: on-chip interconnection networks ». *DAC '01: Proceedings of the 38th conference on Design automation.* Las Vegas, NV. 684–689.

DAVEAU, J.-M., MARCHIORO, G. F., VALDERRAMA, C. A., JERRAYA, A. A. (2002). « Vhdl generation from sdl specifications ». *Readings in hardware/-software co-design.* Kluwer Academic Publishers, Norwell, MA. 125–134.

DESIGN AND REUSE (2004). « SPIRIT consortium drives ip re-use and interoperability with release of specification ». *Design and Reuse.* [En ligne]. http://www.us.design-reuse.com/news/news8024.html (Page consultée le 8 juin 2005)

DOUGLASS, B. P. (2000). *Real-time UML: developing efficient objects for embedded systems.* 2nd ed.

DRECHSLER, R. (2003). « Synthesizing checkers for on-line verification of system-on-chip designs ». *ISCAS '03: Proceedings of the 2003 International Symposium on Circuits and Systems.* 4, 748–751.

DUMAIS, P., AMMARI, M., GAGNON, F., THIBEAULT, C. (2004). « Multi-equalization: a powerful adaptive filtering for time varying wireless channels ». *Proceedings of Vehicular Technology Conference.* Los Angeles, Calif.

EDWARDS, M., GREEN, P. (2003). « 6. UML for hardware and software object modeling ». *UML for Real: Design of Embedded Real-Time Systems.* KAP. 127–147.

GAJSKI, D. D., KUHN, R. H. (1983). « New VLSI tools ». *IEEE Computer.* 16:12. 11–14.

GERSHO, A., LIM, T. (1981). « Adaptive cancellation of intersymbol interference for data transmission ». *Bell System Technical Journal.* 60:11. 1997–2021.

GONZALEZ, R. (2000). « Xtensa: a configurable and extensible processor ». *Micro, IEEE.* 20:2. 60–70.

GUPTA, S., GUPTA, R., DUTT, N., NICOLAU, A. (2004). *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits.*

HAREL, D. (1987). « Statecharts: A visual formalism for complex systems ». *Science of Computer Programming.* 8:3. 231–274.

HAREL, D., NAAMAD, A. (1996). « The STATEMATE semantics of statecharts ». *ACM Trans. Softw. Eng. Methodol.* 5:4. 293–333.

HELAIHEL, R., OLUKOTUN, K. (1997). « Java as a specification language for hardware-software systems ». *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on computer-aided design.* IEEE Computer Society. 690–697.

HELM, R., HOLLAND, I. M., GANGOPADHYAY, D. (1990). « Contracts: Specifying behavioral compositions in object-oriented systems ». *Proc.of the OOPSLA/ECOOP-90.* Ottawa, Canada. 169–180.

HENDERSON-SELLERS, B. (2005). « UML - the good, the bad or the ugly? perspectives from a panel of experts ». *Software and Systems Modeling.* [En ligne]. 4:1. 4–13. http://springerlink.metapress.com/openurl.asp?genre=article&id=doi:10.1007/s10270-004-0076-8 (Page consultée le 8 juin 2005)

HENNESSY, J. L., PATTERSON, D. A. (1996). *Computer Architecture: a Quantitative Approach.* 2nd ed.

INTEL CORPORATION (2005a). *Intel Pentium 4 Processor Datasheet.* [En ligne]. http://www.intel.com/design/pentium4/datashts/298643.htm (Page consultée le 8 juin 2005)

INTEL CORPORATION (2005b). *Moore's Law, the Future - Technology and Research at Intel.* [En ligne]. http://www.intel.com/technology/silicon/mooreslaw/index.htm (Page consultée le 8 juin 2005)

INTERNATIONAL TELECOMMUNICATION UNION (2000). *ITU Recommendation Z.100: Specification and Description Language (SDL).* Geneva.

JACOME, M. F., PEIXOTO, H. P. (2001). « A survey of digital design reuse ». *IEEE Design and Test of Computers.* 98–107.

JANTSCH, A. (2005). « Models of embedded computation ». In Embedded Systems, CRC Press (Invited contribution; to appear).

KEATING, M., BRICAUD, P. (2002). *Reuse methodology manual: for system-on-a-chip designs.* Norwell, MA : Kluwer Academic Publishers.

KEMP, E., PACITTO, D., TODD, E., GRAY, D. (1996). « The role of functional prototyping in model validation ». *Proceedings of the 1996 Information Systems Conference.* IEEE Computer Society Press.

KEUTZER, K., MALIK, S., NEWTON, R., RABAEY, J., SANGIOVANNI-VINCENTELLI, A. L. (2000). « System level design: Orthogonalization of concerns and platform–based design ». *IEEE Trans. on CAD.* 19. 1523–1543.

KUMAR, S., AYLOR, J., JOHNSON, B., WULF, W. (1994). « Object-oriented techniques in hardware design ». *Computer.* 27:6. 64–70.

LAMMERS, D. (2005). « Shift to 65-nm designs likely to be quick ». *EE-Times.* [En ligne]. www.eetimes.com/news/design/technology/showArticle.jhtml?articleID=160503281 (Page consultée le 8 juin 2005)

LAPALME, J., ABOULHAMID, E., NICOLESCU, G., CHAREST, L., BOYER, F., DAVID, J., BOIS, G. (2004). « .NET framework - a solution for the next generation tools for system-level modeling and simulation ». *Proceedings of the Design, Automation and Test in Europe Conference.*

LAVAGNO, L., MARTIN, G., SELIC, B. (2003). *UML for Real: Design of Embedded Real-Time Systems.*

LEV, L., RAZDAN, R., TICE, C. (2003). « It's about time : Requirements for the functional verfification of nanometer-scale ics ». Technical report, Cadence Design Systems Inc.

LI, Y.-T. S., MALIK, S. (1999). *Performance Analysis of Real-Time Embedded Software.*

MAME (2005). *Méthodologies et architectures pour la multiégalisation.* [En ligne]. http://www.ele.etsmtl.ca/projets/PROMPT/ACCUEIL.htm (Page consultée le 8 juin 2005)

MARTIN, G., CHANG, H. (2003). *Winning the SoC Revolution - Experiences in Real Design.*

MARTIN, G., LAVAGNO, L., LOUIS-GUERIN, J. (2001). « Embedded UML: a merger of real-time UML and co-design ». *Proceedings of the Ninth International Symposium on Hardware-Software Codesign (CODES).* 23–28.

MCFARLAND, M., PARKER, A., CAMPOSANO, R. (1990). « The high-level synthesis of digital systems ». *Proceedings of the IEEE.* 78:2. 301–318.

MENTOR GRAPHICS CORPORATION (2005). *Catapult-C synthesis.* [Logiciel]. Wilsonville, Oregon.

MEYER, B. (1997). *Object-Oriented Software Construction.* 2 ed. Upper Saddle River, NJ : Prentice Hall.

MILLER, J., MUKERJI, J. (2003). *MDA Guide, Version 1.0.1.* [En ligne]. Object Management Group. http://www.omg.org/docs/omg/03-06-01.pdf (Page consultée le 8 juin 2005)

MITOLA, J. (2000). *Software Radio Architecture: Object Oriented Approaches to.*

MOTUS, L. (2003). « 10. modeling metric time ». *UML for Real: Design of Embedded Real-Time Systems.* KAP. 205–220.

NICOLESCU, G. (2002). *Spécification et validation des systèmes hétérogènes embarqués.* Thèse de doctorat, Institut national polytechnique de Grenoble.

OBJECT MANAGEMENT GROUP (2002). *UML Profile for Schedulability, Performance and Time Specification.* [En ligne]. http://www.omg.org/cgi-bin/doc?formal/2005-01-02 (Page consultée le 8 juin 2005)

OBJECT MANAGEMENT GROUP (2004a). *The Common Object Request Broker: Architecture and Specification.* [En ligne], 3.0.3 ed. http://www.omg.org/technology/documents/formal/corba_iiop.htm (Page consultée le 8 juin 2005)

OBJECT MANAGEMENT GROUP (2004b). *UML 2.0 Superstructure Specification (Working Document).* [En ligne]. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02 (Page consultée le 8 juin 2005)

OPENCORES (2002). *Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores.* [En ligne], b.3 ed. http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf (Page consultée le 8 juin 2005)

OSCI (2003). *SystemC 2.0.1 Language Reference Manual, Revision 1.0.*

PAULIN, P., PILKINGTON, C., BENSOUDANE, E. (2002). « StepNP: A system-level exploration platform for network processors ». *IEEE Design and Test.* 19:6. 17–26.

PROAKIS, J. G. (2000). *Digital Communications.* 4th ed.

QURESHI, S. U. H. (1985). « Adaptive equalization ». *Proceedings of the IEEE.* 53, 1349–1387.

RATIONAL SOFTWARE CORPORATION (2002). *UML-RT language guide, Rational Rose software documentation.*

RATIONAL SOFTWARE CORPORATION (2003). *Rational Rose Real-Time, ver. 2003.06.00.436.000.* [Logiciel]. Cupertino, Calif.

ROWSON, J. A., SANGIOVANNI-VINCENTELLI, A. (1997). « Interface-based design ». *Proceedings of the 34th Design Automation Conference.* ACM Press. 178–183.

SANGIOVANNI-VINCENTELLI, A., MARTIN, G. (2001). « Platform-based design and software design methodology for embedded systems ». *IEEE Design and Test of Computers.* 18:6. 23–33.

SCIUTO, D., MARTIN, G., ROSENSTIEL, W., SWAN, S., GHENASSIA, F., FLAKE, P., SROUJI, J. (2004). « SystemC and SystemVerilog: Where do they fit? where are they going? ». *DATE '04: Proceedings of the conference on Design, automation and test in Europe.* Washington, DC, IEEE Computer Society. 10122.

SELIC, B., GULLEKSON, G., WARD, P. (1994). *Real-Time Object Oriented Modeling.*

SELIC, B., RUMBAUGH, J. (1998). « Using UML for modeling complex real-time systems ». *White paper, Rational and ObjectTime.*

SEMICONDUCTOR INDUSTRY ASSOCIATION (1999). *International Technology Roadmap for Semiconductors (ITRS).* [En ligne]. http://public.itrs.net/ (Page consultée le 8 juin 2005)

SEMICONDUCTOR INDUSTRY ASSOCIATION (2004). *International Technology Roadmap for Semiconductors (ITRS).* [En ligne]. http://public.itrs.net/ (Page consultée le 8 juin 2005)

TELELOGIC (2005). *Tau SDL suite.* [Logiciel]. Malmö, Sweden.

THE MATHWORKS (2005). *Simulink 7.0.* [Logiciel]. Natick, Mass.

VINCENTELLI, A. S. (2002). « Defining platform-based design ». *EEDesign of EETimes.* [En ligne]. http://www.gigascale.org/pubs/141.html (Page consultée le 8 juin 2005)

VSI ALLIANCE (2005). *Page d'accueil.* [En ligne]. http://www.vsi.org/ (Page consultée le 8 juin 2005)

WILSON, R. (2003). « OCP, VSIA join forces for SoC interconnect ». *EETimes.* [En ligne]. http://www.eetimes.com/story/OEG20031007S0017 (Page consultée le 8 juin 2005)

WINSKEL, G., NIELSEN, M. (1995). *Models for Concurrency.*

ZHU, Q., MATSUDA, A., KUWAMURA, S., NAKATA, T., SHOJI, M. (2002). « An object-oriented design process for system-on-chip using uml ». *International Symposium on System Synthesis.* 249–254.

ANNEXE I

DIAGRAMMES UML

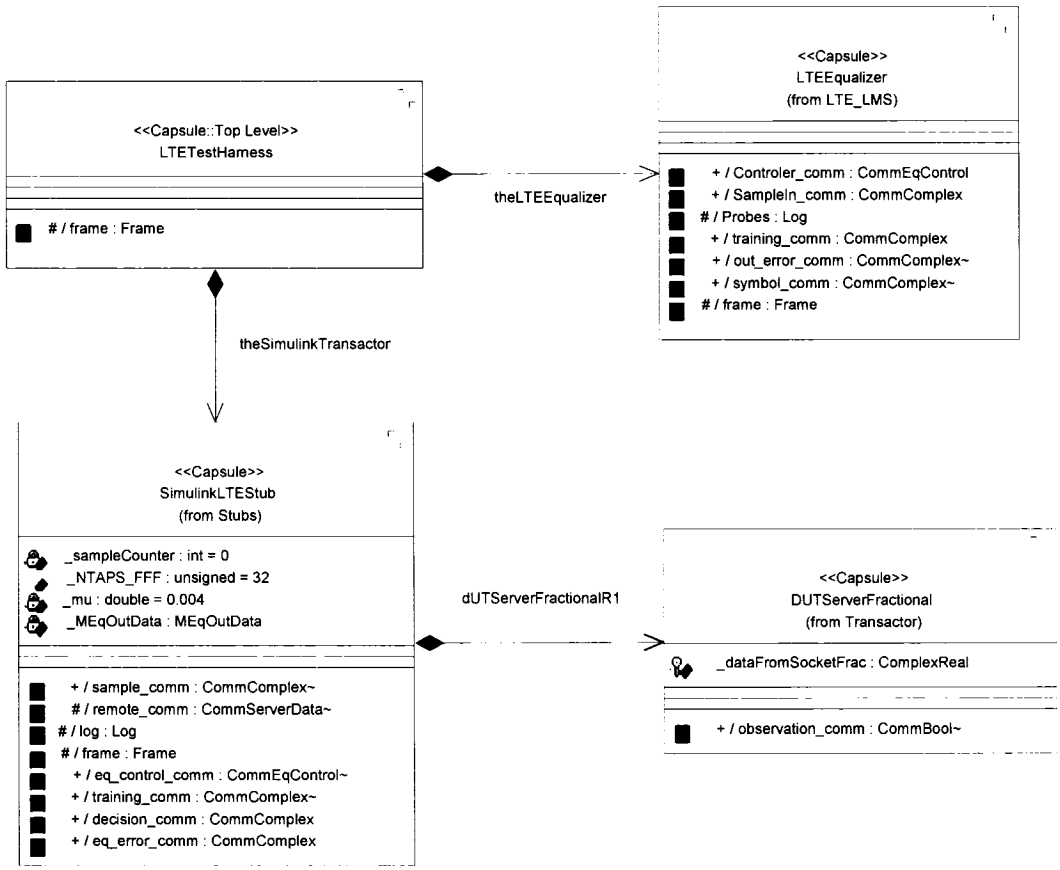

FIG. I.1: Diagramme des classes qui composent la capsule principale

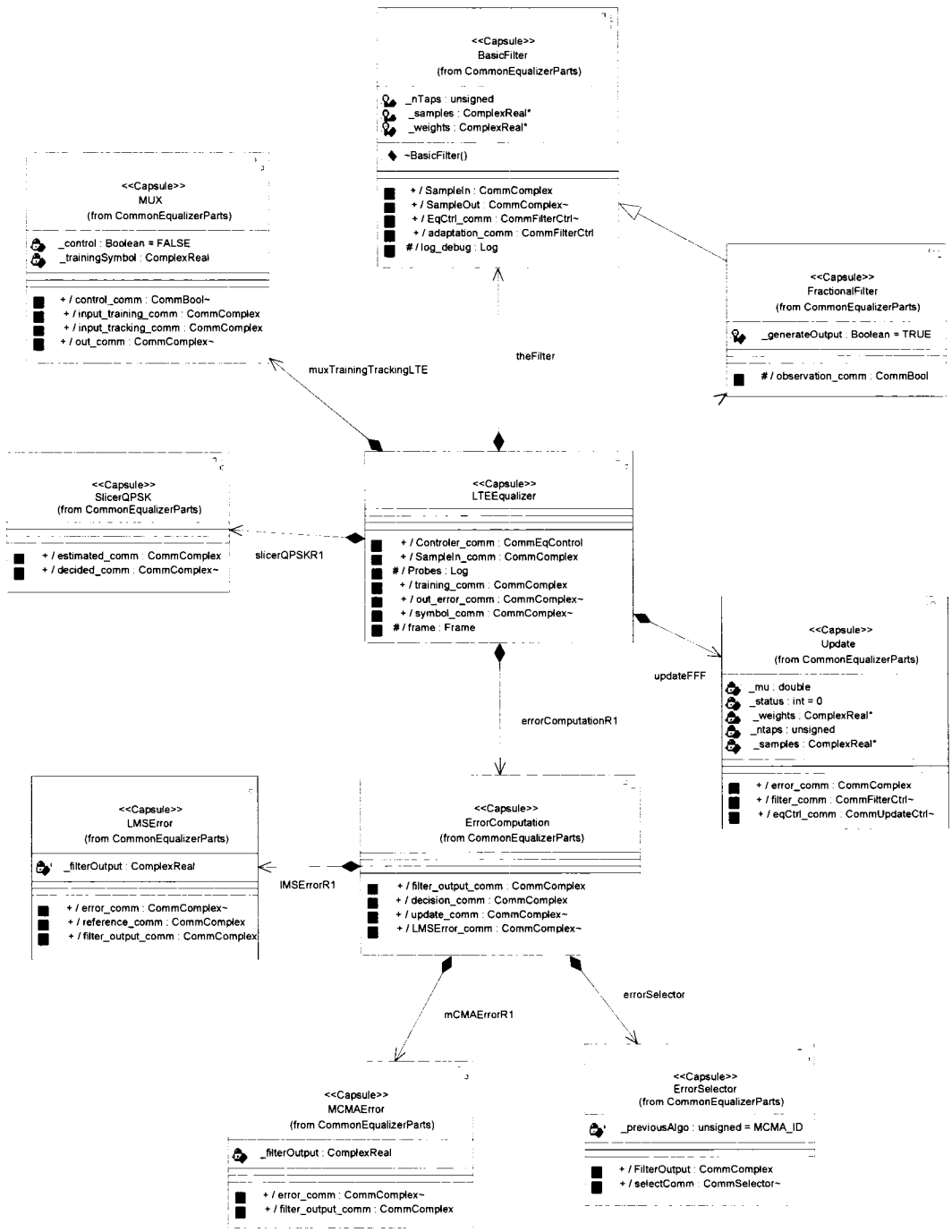

FIG. I.2: Diagramme des classes qui composent l'égaliseur adaptatif linéaire

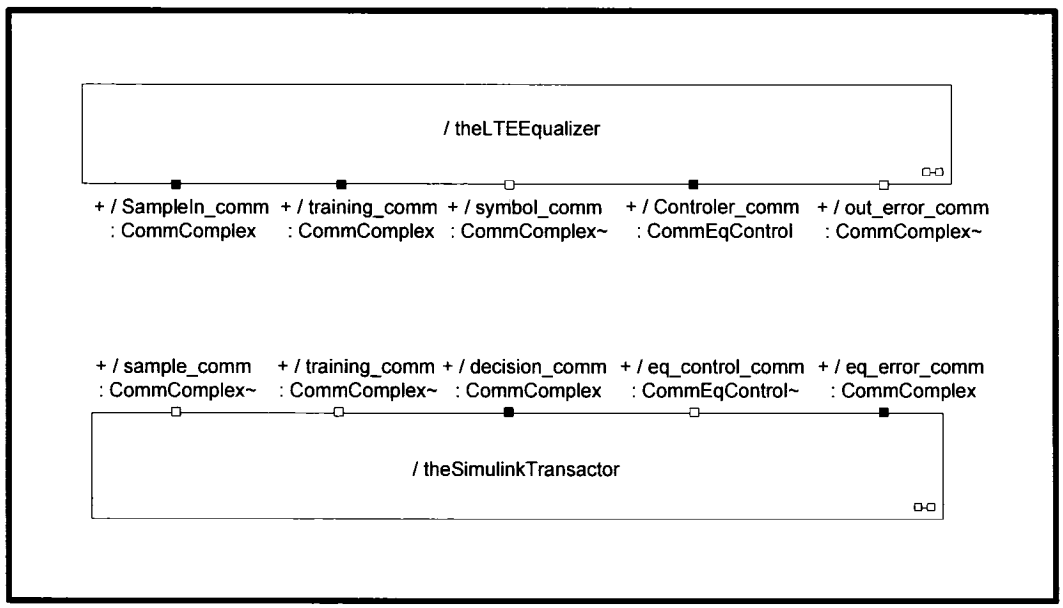

FIG. I.3: Diagramme de structure de la capsule principale

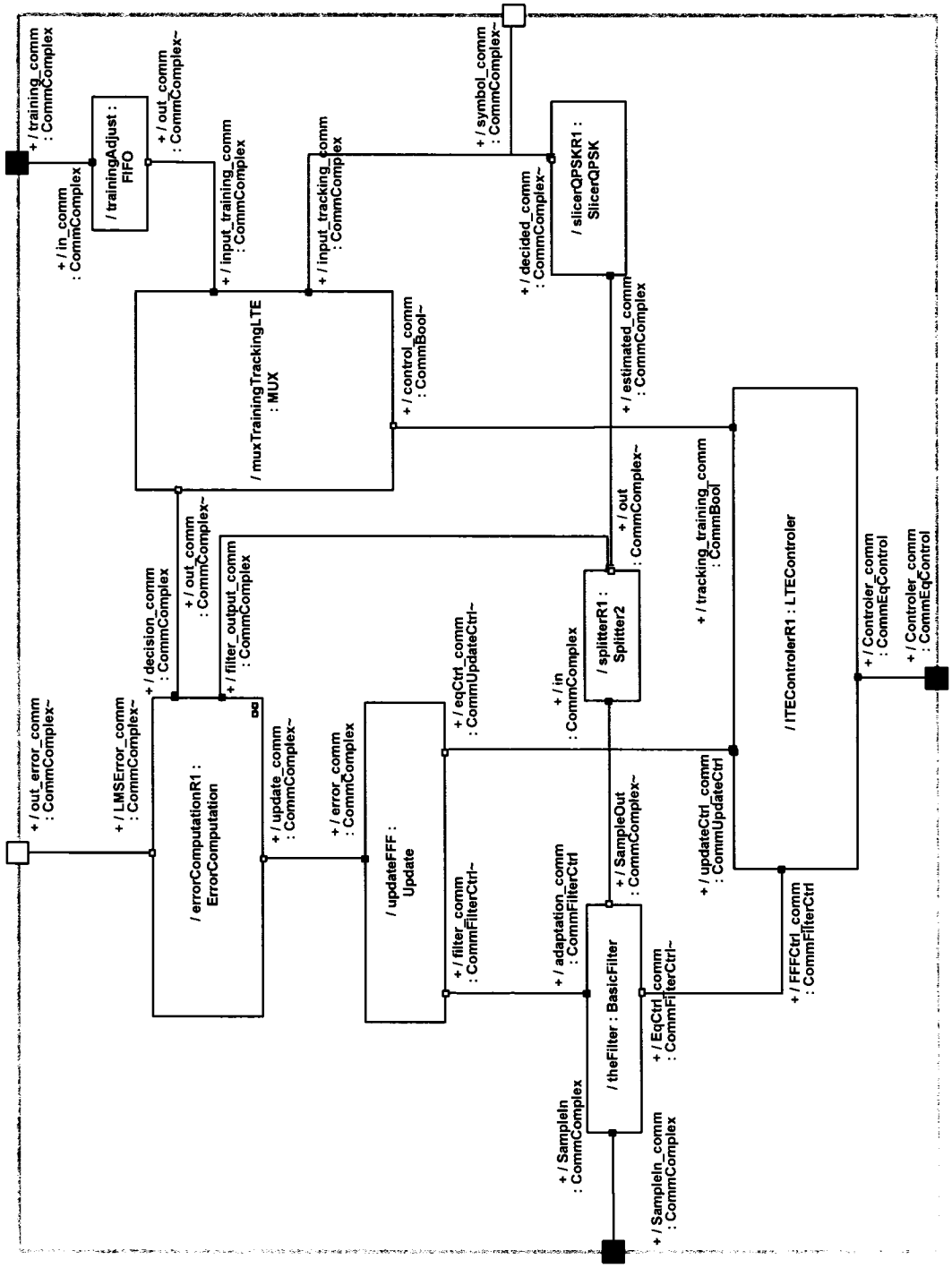

FIG. I.4: Diagramme de structure de l'égaliseur

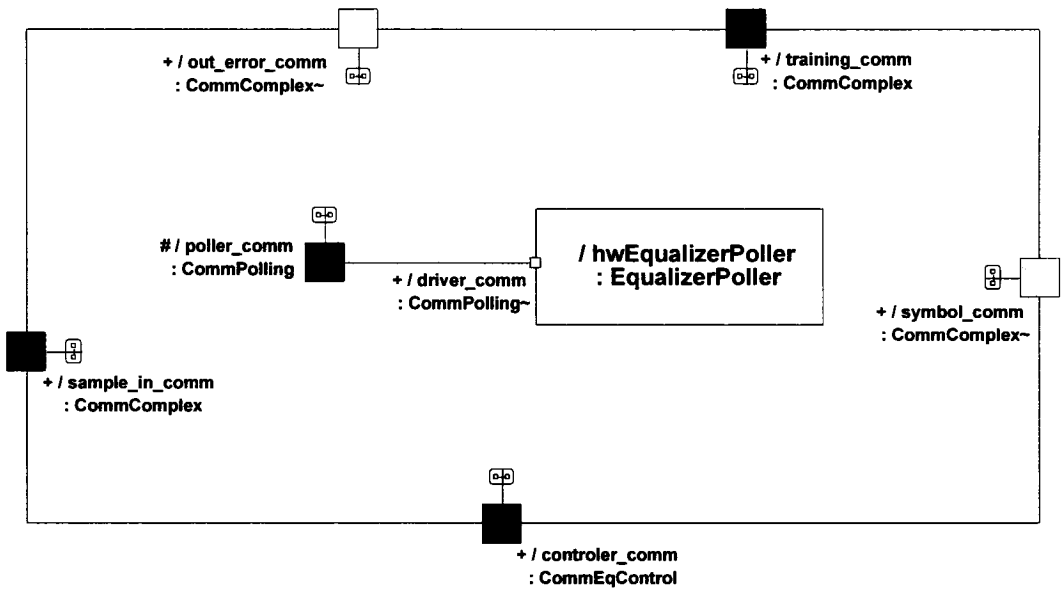

FIG. I.5: Diagramme de structure du pilote de l'égaliseur

ANNEXE II

CODE DE LA S-FUNCTION DE TRANSACTION

Cette annexe présente le code source du bloc de type S-Function qui constitue la moitié client du transactor de niveau modèle.

Listage II.1: DUTClientFrac.cpp

```
/*****************************************************************************
 * File    : DUTClientFrac.cpp
 *
 * Author : A. Chureau
 *
 * Description : Client stub of the LTE
 *
 * Input parameters :
 *   IP address of the server. Example : '148.124.12.73'
 *   Port on the server. Example : 27015
 *
 * Comments :
 *   This is the S-Function code of the Model-Level transactor. It implements
 *   a client connection to a server. Use S-Function parameters to configure
 *   the IP address and port of the server.
 *
 * To do :
 *
 * Revision history :
 * £Log: DUTClientFrac.cpp,v £
 * Revision 1.1  2005/04/01 23:27:08  alexchu
 * Configuré la SFunction fractionnelle du linear_eq_lms_frac_socket.mdl
 *
 * Revision 1.12  2005/02/07 18:36:10  alexchu
 * Nettoyage : ajout de ifdef DEBUG etc.
 *
 * Revision 1.11  2005/01/11 22:38:22  alexchu
 * Ajout de la configuration IP par paramètres (adresse + port) : double-click
 * sur la S-Function pour configurer.
 *
 *****************************************************************************/

#include <stdio.h>
#include <time.h>      //For delay mechanism using clock()
#include "winsock2.h" //To use socket connections
#include "matrix.h"    //To use mxArray manipulation functions

//============================================================================
// Include common data structure here
#ifdef VC6
    #include "/work/src_mame/uml/VC6_LMSEqualizerComponent/src/ComplexReal.h"
      * //Complex data class defined in UML
    #include "/work/src_mame/uml/VC6_LMSEqualizerComponent/src/EqInData.h" //
        Equalizer data class defined in UML
    #include "/work/src_mame/uml/VC6_LMSEqualizerComponent/src/MEqOutData.h" //
        Equalizer data class defined in UML
```

```
44  #else
45      #include "/work/src_mame/uml/VC7_LTEEqualizerComponent/src/ComplexReal.h"
            //Complex data class defined in UML
46      #include "/work/src_mame/uml/VC7_LTEEqualizerComponent/src/EqInData.h" //
            Equalizer data class defined in UML
47      #include "/work/src_mame/uml/VC7_LTEEqualizerComponent/src/MEqOutData.h" //
            Equalizer data class defined in UML
48  #endif
49  //=============================================================================
50
51  extern "C" { // use the C fcn-call standard for all functions
52               // defined within this scope
53
54  #define S_FUNCTION_LEVEL 2
55  #define S_FUNCTION_NAME  DUTClientFrac
56
57  #include "simstruc.h" //Definition of SimStruct and its associated macro
        definitions.
58
59  #define NUM_INPUTS          3
60
61  /* Input Port  0 */
62  #define IN_PORT_0_NAME      u0
63  #define INPUT_0_WIDTH       1
64  #define INPUT_DIMS_0_COL    1
65  #define INPUT_0_DTYPE       creal_T
66  #define INPUT_0_COMPLEX     COMPLEX_YES
67  #define IN_0_FRAME_BASED    FRAME_NO
68  #define IN_0_DIMS           1-D
69  #define INPUT_0_FEEDTHROUGH 1
70
71  /* Input Port  1 */
72  #define IN_PORT_1_NAME      training
73  #define INPUT_1_WIDTH       1
74  #define INPUT_DIMS_1_COL    1
75  #define INPUT_1_DTYPE       creal_T
76  #define INPUT_1_COMPLEX     COMPLEX_YES
77  #define IN_1_FRAME_BASED    FRAME_NO
78  #define IN_1_DIMS           1-D
79  #define INPUT_1_FEEDTHROUGH 1
80
81  /* Input Port  2 */
82  #define IN_PORT_2_NAME      mode
83  #define INPUT_2_WIDTH       1
84  #define INPUT_DIMS_2_COL    1
85  #define INPUT_2_DTYPE       boolean_T
86  #define INPUT_2_COMPLEX     COMPLEX_NO
87  #define IN_2_FRAME_BASED    FRAME_NO
88  #define IN_2_DIMS           1-D
89  #define INPUT_2_FEEDTHROUGH 1
90
91
92  #define NUM_OUTPUTS          1
93
94  /* Output Port 0 */
95  #define OUT_PORT_0_NAME      y0
96  #define OUTPUT_0_WIDTH       1
97  #define OUTPUT_DIMS_0_COL    1
98  #define OUTPUT_0_DTYPE       creal_T
99  #define OUTPUT_0_COMPLEX     COMPLEX_YES
100 #define OUT_0_FRAME_BASED    FRAME_NO
101 #define OUT_0_DIMS           1-D
102
103 // Some other useful defines
104 #define MAIN_CYCLE 0         // Used to toggle between main and fractional
        cycles
105 #define FRACTIONAL_CYCLE 1
106 #define OFFSET 1             // 0 to produce a symbol at the 2nd input symbol
```

```
107                                  // 1            "              1st     "
108
109  //Function prototype for this S-Function
110  void send_data(SimStruct *S, MEqOutData &result);
111  SOCKET call_socket(const char *hostname, unsigned short portnum);
112
113  /* Function: call_socket ================================================
114   * Abstract:
115   *
116   *
117   *
118   */
119  SOCKET call_socket(const char *hostname, unsigned short portnum) {
120    struct sockaddr_in sa;
121    SOCKET s;
122
123    //  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
124    s = socket(AF_INET, SOCK_STREAM, 0);
125    if (s == INVALID_SOCKET)
126      return INVALID_SOCKET;
127
128    /* set address */
129    sa.sin_family = AF_INET;
130    sa.sin_addr.s_addr = inet_addr(hostname);
131    //sa.sin_addr.s_addr = inet_addr("142.137.20.130");
132    sa.sin_port = htons(portnum);
133
134    /* try to connect to the specified socket */
135    if (connect(s, (SOCKADDR*) &sa, sizeof(sa)) == SOCKET_ERROR) {
136      closesocket(s);
137      return INVALID_SOCKET;
138    }
139    return s;
140  }
141
142  /*====================*
143   * S-function methods *
144   *====================*/
145
146  /* Function: mdlInitializeSizes =========================================
147   * Abstract:
148   *    The sizes information is used by Simulink to determine the S-function
149   *    block's characteristics (number of inputs, outputs, states, etc.).
150   */
151  static void mdlInitializeSizes(SimStruct *S)
152  {
153      /* See sfuntmpl_doc.c for more details on the macros below */
154
155      ssSetNumSFcnParams(S, 2);  /* Number of expected parameters */
156      if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
157          /* Return if number of expected != number of actual parameters */
158          return;
159      }
160
161      //| The S-function needs persistent storage for
162      //|    -The socket that is opened to communicate with the design
163      //|    -The state of the signal being transferred (fractional or not)
164      //| Reserve elements in a work vector for this purpose
165      //| (c.f. online Simulink documentation)
166      //Reserve element to store a persistent socket as an integer
167      ssSetNumIWork(S, 2);
168
169      ssSetNumContStates(S, 0);
170      ssSetNumDiscStates(S, 0);
171
172      if (!ssSetNumInputPorts(S, NUM_INPUTS)) return;
173
174      /*Input Port 0 : sample value */
```

```
175     ssSetInputPortWidth(S,  0, INPUT_0_WIDTH); /* */
176     ssSetInputPortDataType(S,  0, SS_DOUBLE);
177     ssSetInputPortComplexSignal(S,  0, INPUT_0_COMPLEX);
178     ssSetInputPortDirectFeedThrough(S, 0, INPUT_0_FEEDTHROUGH);
179     ssSetInputPortRequiredContiguous(S, 0, 1); /*direct input signal access*/
180
181     /*Input Port 1 : training symbol */
182     ssSetInputPortWidth(S,  1, INPUT_1_WIDTH); /* */
183     ssSetInputPortDataType(S,  1, SS_DOUBLE);
184     ssSetInputPortComplexSignal(S,  1, INPUT_1_COMPLEX);
185     ssSetInputPortDirectFeedThrough(S, 1, INPUT_1_FEEDTHROUGH);
186     ssSetInputPortRequiredContiguous(S, 1, 1); /*direct input signal access*/
187
188     /*Input Port 2 : training mode (true = training, false = tracking) */
189     ssSetInputPortWidth(S,  2, INPUT_2_WIDTH); /* */
190     ssSetInputPortDataType(S,  2, SS_BOOLEAN);
191     ssSetInputPortComplexSignal(S,  2, INPUT_2_COMPLEX);
192     ssSetInputPortDirectFeedThrough(S, 2, INPUT_2_FEEDTHROUGH);
193     ssSetInputPortRequiredContiguous(S, 2, 1); /*direct input signal access*/
194
195     if (!ssSetNumOutputPorts(S, NUM_OUTPUTS)) return;
196
197     /*Output Port 0 */
198     ssSetOutputPortWidth(S, 0, OUTPUT_0_WIDTH);
199     ssSetOutputPortDataType(S, 0, SS_DOUBLE);
200     ssSetOutputPortComplexSignal(S, 0, OUTPUT_0_COMPLEX);
201
202
203     ssSetNumSampleTimes(S, 1);
204     ssSetOptions(S, 0);
205 }
206
207
208
209
210 /* Function: mdlInitializeSampleTimes =========================================
211  * Abstract:
212  *    This function is used to specify the sample time(s) for your
213  *    S-function. You must register the same number of sample times as
214  *    specified in ssSetNumSampleTimes.
215  */
216 static void mdlInitializeSampleTimes(SimStruct *S)
217 {
218     ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
219     ssSetOffsetTime(S, 0, 0.0);
220
221 }
222
223 #define MDL_START  /* Change to #undef to remove function */
224 #if defined(MDL_START)
225   /* Function: mdlStart =======================================================
226    * Abstract:
227    *    This function is called once at start of model execution. If you
228    *    have states that should be initialized once, this is the place
229    *    to do it.
230    */
231   static void mdlStart(SimStruct *S)
232   {
233   WSADATA info;
234   SOCKET sock;
235   char *ipAddressBuf;
236   int addrBufLen;
237   unsigned short portNum;
238   char tbuffer [9];
239
240   // Retrieve the IP address of the target module from the parameter dialog box
241   // The address should be passed as a string
```

```
242     addrBufLen = (mxGetM(ssGetSFcnParam(S, 0)) * mxGetN(ssGetSFcnParam(S, 0)))
            + 1;
243     ipAddressBuf = (char *) mxCalloc(addrBufLen, sizeof(char));
244     mxGetString(ssGetSFcnParam(S, 0), ipAddressBuf, addrBufLen);
245
246     // Now retrieve the port of the target. Should be passed as an int.
247     portNum = (unsigned short) mxGetScalar(ssGetSFcnParam(S, 1));
248
249     if (WSAStartup(MAKEWORD(2,2), &info) == SOCKET_ERROR){
250       ssPrintf("Could not initalize socket library");
251       return;
252       }
253
254     // Call the socket initialization routine
255     sock = call_socket(ipAddressBuf, portNum);
256     if (sock == INVALID_SOCKET) {
257       ssPrintf("Could not connect");
258       return;
259       }
260
261     //Store socket in persistent vector so we can use it later
262     ssSetIWorkValue(S,0,sock);
263
264     //Print start time (for profiling purpose)
265     _strtime( tbuffer );
266     ssPrintf( "Start time is %s \n", tbuffer );
267     }
268
269   #endif /*  MDL_START */
270
271   // This function enables fractionnally spaced equalization
272   #define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
273   #if defined(MDL_INITIALIZE_CONDITIONS)
274     /* Function: mdlInitializeConditions ========================================
275      * Abstract:
276      *    In this function, you should initialize the continuous and discrete
277      *    states for your S-function block.  The initial states are placed
278      *    in the state vector, ssGetContStates(S) or ssGetDiscStates(S).
279      *    You can also perform any other initialization activities that your
280      *    S-function may require. Note, this method will be called at the
281      *    start of simulation and if it is present in an enabled subsystem
282      *    configured to reset states, it will be call when the enabled subsystem
283      *    restarts execution to reset the states.
284      *
285      *    You can use the ssIsFirstInitCond(S) macro to determine if this is
286      *    is the first time mdlInitializeConditions is being called.
287      */
288     static void mdlInitializeConditions(SimStruct *S)
289     {
290         // Set initial state for fractionnaly-spaced equalizers
291         if (OFFSET == 1){
292             ssSetIWorkValue(S, 1, FRACTIONAL_CYCLE);
293
294             //Wait here for the first REQ from the DUV
295             SOCKET sock = (SOCKET) ssGetIWorkValue(S,0);  //Retrieve socket from
                    work vector
296             if (sock == INVALID_SOCKET) {
297               ssSetErrorStatus(S, "Invalid socket (mdlInitializeConditions)");
298               return;
299             }
300
301             int bytesRecv = SOCKET_ERROR;
302             char recvBuf[32] = "";
303
304             while (bytesRecv == SOCKET_ERROR){
305                 bytesRecv = recv(sock, recvBuf, 32, 0);
306                 if (bytesRecv == 0 || bytesRecv == WSAECONNRESET) {
307                     ssWarning(S, "Connection Closed");
```

```
308                     ssSetStopRequested(S,1);
309                     break;
310                 }
311                 if (bytesRecv < 0){
312                     ssWarning(S, "Problem with connection. The simulation will
                        stop");
313                     ssSetStopRequested(S,1);
314                     break;
315                 }
316             } //while (bytesRecv == SOCKET_ERROR)
317
318             //Leave if an error occured
319             if (ssGetStopRequested(S))
320                 return;
321
322             #ifdef DEBUG_SFUNCTION
323                 ssPrintf("Received Request during INITIALIZATION --- Bytes
                        Received: %d\n", bytesRecv);
324             #endif
325
326         }
327         else
328             ssSetIWorkValue(S, 1, MAIN_CYCLE);
329     }
330 #endif /* MDL_INITIALIZE_CONDITIONS */
331
332
333 /* Function: mdlOutputs =======================================================
334  * Abstract:
335  *     In this function, you compute the outputs of your S-function
336  *     block. Generally outputs are placed in the output vector, ssGetY(S).
337  */
338 static void mdlOutputs(SimStruct *S, int_T tid)
339 {
340   MEqOutData resultFromEq; //To store the result from the equalizer
341
342   send_data(S, resultFromEq);
343
344   creal64_T *y   = (creal64_T *)ssGetOutputPortSignal(S,0);
345
346   //Width of signal is 1, therefore there is only 1 element in the array (index
        0)
347   y[0].re = resultFromEq.error_eq1.re;
348   y[0].im = resultFromEq.error_eq1.im;
349   #ifdef DEBUG_SFUNCTION
350     ssPrintf("Result Received : re=%f im=%f\n", y[0].re, y[0].im);
351   #endif
352 }
353
354 /* Function: mdlTerminate =====================================================
355  * Abstract:
356  *     In this function, you should perform any actions that are necessary
357  *     at the termination of a simulation.  For example, if memory was
358  *     allocated in mdlStart, this is the place to free it.
359  */
360
361 static void mdlTerminate(SimStruct *S)
362 {
363   char tbuffer [9];
364   //Disconnect socket
365   WSACleanup();
366   //Print OS time for profiling
367   _strtime( tbuffer );
368   ssPrintf( "End time is %s \n", tbuffer );
369 }
370
371 /*=============================================================*
372  * See sfuntmpl_doc.c for the optional S-function methods *
```

```
373   *=========================================================*/
374
375
376  /* Function: send_data ====================================================
377   * Abstract: main communication function with UML : sends data to the DUV
378   *
379   */
380
381  //ComplexReal send_data(SimStruct *S){
382  void send_data(SimStruct *S, MEqOutData &result){
383    SOCKET sock; //Used for socket communication. It is retrieved from the static
           work space of the SimStruct.
384    EqInData dataOut; //Data exchange structure used for communication over
           socket
385    MEqOutData resultData; //Data computed by equalizer and returned to this
           function over socket
386    int_T currentCycle; // The state of the current cycle
387
388    // Check if function was invoked during fractional or main cycle
389    currentCycle = ssGetIWorkValue(S, 1);
390
391    //Retrieve input values
392    const creal64_T *u0          = (const creal64_T*) ssGetInputPortSignal(S,0);
393    const creal64_T *trainSymbol = (const creal64_T*) ssGetInputPortSignal(S,1);
394    const boolean_T *trainMode = (const boolean_T*) ssGetInputPortSignal(S,2);
395
396    //Initialization of data exchange structure
397    // Sample
398    dataOut.in.re = (double) (*u0).re;
399    dataOut.in.im = (double) (*u0).im;
400    // Training symbol
401    dataOut.inTraining.re = (double) (*trainSymbol).re;
402    dataOut.inTraining.im = (double) (*trainSymbol).im;
403    // Training mode
404    dataOut.mode  = (bool) (*trainMode);
405
406    //Initilization of output signal in case something goes wrong before function
           completes
407    resultData.error_eq1.re       = 0.0;
408    resultData.error_eq1.im       = 0.0;
409    resultData.error_eq2.re       = 0.0;
410    resultData.error_eq2.im       = 0.0;
411    resultData.manager_selection  = 0;
412    resultData.MEqSymbol.re       = 0.0;
413    resultData.MEqSymbol.im       = 0.0;
414    resultData.mse_eq1            = 0.0;
415    resultData.mse_eq2            = 0.0;
416    resultData.mse_min            = 0.0;
417
418    // CLEAN THIS EVENTUALLY, result should be replaced by resultData
419    result = resultData;
420
421    //Could eventually use this for error logging
422    //ofstream outFile("c:/test_socket_cpp.txt", ios::app);
423    //outFile << "Waiting for request\n";
424
425    //Retrieve socket from work vector
426    sock = (SOCKET) ssGetIWorkValue(S,0);
427
428    int bytesSent;
429    int bytesRecv = SOCKET_ERROR;
430    char sendBuf[32] = "Client: salut patate";
431    char recvBuf[32] = "";
432
433    // During MAIN_CYCLE state : wait for the synchronization signal from UML
434    if (currentCycle == MAIN_CYCLE){
435        while (bytesRecv == SOCKET_ERROR){
436            bytesRecv = recv(sock, recvBuf, 32, 0);
```

```
437             if (bytesRecv == 0 || bytesRecv == WSAECONNRESET) {
438                 ssWarning(S, "Connection Closed");
439                 ssSetStopRequested(S,1);
440                 break;
441             }
442             if (bytesRecv < 0){
443                 ssWarning(S, "Problem with connection. The simulation will stop")
                        ;
444                 ssSetStopRequested(S,1);
445                 break;
446             }
447         } //while (bytesRecv == SOCKET_ERROR)
448
449         //Leave if an error occured
450         if (ssGetStopRequested(S))
451             return;
452
453         #ifdef DEBUG_SFUNCTION
454             ssPrintf("Received Request --- Bytes Received: %d\n", bytesRecv);
455         #endif
456     } //if (currentCycle == MAIN_CYCLE)
457
458
459     //Send data to device under test over socket
460     bytesSent = send(sock, (char*) &dataOut, sizeof(dataOut), 0);
461     #ifdef DEBUG_SFUNCTION
462         ssPrintf("Sent RE=%f IM=%f --- Bytes Sent: %ld\n", dataOut.in.re, dataOut
                .in.im, bytesSent);
463         ssPrintf("Hold         = %ld\n", sizeof(dataOut.hold));
464         ssPrintf("in           = %ld\n", sizeof(dataOut.in));
465         ssPrintf("inTraining   = %ld\n", sizeof(dataOut.inTraining));
466         ssPrintf("mode         = %ld\n", sizeof(dataOut.mode));
467         ssPrintf("mu           = %ld\n", sizeof(dataOut.mu));
468         ssPrintf("reset        = %ld\n", sizeof(dataOut.reset));
469     #endif
470
471      // Stop here if we are in a MAIN_CYCLE
472     if (currentCycle == MAIN_CYCLE)
473         return;
474
475     //| ------------------------------------------------------------------
476     //| SECOND PART of the transaction : receiving processed data from
477     //| the UML model. We should reach this code only when in state
478     //| FRACTIONAL_CYCLE.
479     //| ------------------------------------------------------------------
480
481     //Wait for processed data to come back
482     bytesRecv = SOCKET_ERROR;
483     char resultBuf[100] = "";
484
485     while (bytesRecv == SOCKET_ERROR){
486
487       //Only the required number of bytes is read : sizeof(resultData), usually
              16 bytes.
488       //This is necessary because a request is sent following the error.
489       //The request bytes may be concatenated with the error bytes if Simulink
490       //is not fast enough to read the error on the socket before the request is
              sent.
491
492       bytesRecv = recv(sock, resultBuf, sizeof(resultData), 0);
493
494       if (bytesRecv == 0 || bytesRecv == WSAECONNRESET) {
495         ssPrintf("Connection Closed\n");
496         ssSetStopRequested(S,1);
497         break;
498       }
499       if (bytesRecv != sizeof(resultData)){
```

```
500         ssPrintf("Error on recv of computed error. Expecting %ld bytes, received
                %ld\n",
501                  sizeof(resultData), bytesRecv);
502         ssSetStopRequested(S,1);
503         break;
504         }
505       else{
506         memcpy(&resultData, resultBuf, bytesRecv);
507         #ifdef DEBUG_SFUNCTION
508             ssPrintf("Result Received : re=%f im=%f, NumBytes = %d\n", resultData
                    .error_eq1.re, resultData.error_eq1.im, bytesRecv);
509         #endif
510       }
511     }
512
513     result = resultData;
514   }
515
516   #define MDL_UPDATE  /* Change to #undef to remove function */
517   #if defined(MDL_UPDATE)
518     /* Function: mdlUpdate ======================================================
519      * Abstract:
520      *    This function is called once for every major integration time step.
521      *    Discrete states are typically updated here, but this function is useful
522      *    for performing any tasks that should only take place once per
523      *    integration step.
524      */
525     static void mdlUpdate(SimStruct *S, int_T tid)
526     {
527     //| Update the state of the block : will the next cycle send a "main" or "
            fractional"
528     //| symbol to the design. In the case of 2-fractionally-spaced equalizers,
            this
529     //| corresponds to a simple toggling of the state. The state is taken into
530     //| account during the next cycle.
531
532     int_T currentCycle = ssGetIWorkValue(S, 1); //The value of the current cycle
533
534     if (currentCycle == MAIN_CYCLE)
535       ssSetIWorkValue(S, 1, FRACTIONAL_CYCLE);
536     else
537       ssSetIWorkValue(S, 1, MAIN_CYCLE);
538     }
539   #endif /* MDL_UPDATE */
540
541   /*=============================*
542    * Required S-function trailer *
543    *=============================*/
544
545   #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
546   #include "simulink.c"      /* MEX-file interface mechanism */
547   #else
548   #include "cg_sfun.h"       /* Code generation registration function */
549   #endif
550
551   //#ifdef __cplusplus
552   } // end of extern "C" scope
553   //#endif
```

ANNEXE III

SCRIPT DE TRANSFORMATION D'UN MODÈLE UML RATIONAL ROSE EN UN MODÈLE SIMULINK

Listage III.1: Fichier simulinkGen.ebs

```
'------------------------------------------------------------------------------
'simulinkGen.ebs
'
'   Generates Simulink model files from structure diagrams.
'
'   Script Requires:
'       Installed version of Matlab 6.5
'       Active Structure Diagram in Rose-RT
'
'   Required Files:
'
' Created by Alexandre Chureau
' (C) Ecole Polytechnique de Montreal 2004   All Rights Reserved
'------------------------------------------------------------------------------
' $Source: /home/cvs/src_mame/uml/script/simulinkGen.ebs,v $
' $Log: simulinkGen.ebs,v $
' Revision 1.6  2004/03/02 17:38:09  alexchu
' Changed name of S-function to name of capsule role
'
' Revision 1.4  2004/02/27 22:53:51  alexchu
' Fixed documentation bugs after core meeting of feb 26
' Created CheckCommand
'
' Revision 1.3  2004/02/26 02:22:24  alexchu
' Done with in/out ports of S-function builder blocks. Still have to connect
' the blocks to the rest of the model.
'
'------------------------------------------------------------------------------

Const MFILE_EXT_1 = "_cfg"
Const MFILE_EXT = MFILE_EXT_1 & ".m"
Const CRLF$ = Chr$(13) + Chr$(10)
Const SCALE% = 3.0
Const FATAL = 0
Const NORMAL = 1
Const PORT_IN = 0 'Port directions
Const PORT_OUT = 1

'OLE Automation object available to all subs & func of this script
Private matlabServer As Object

'The global M-File name
Private globalMFileName As String

'------------------------------------------------------------------
' This subroutine checks a command's returned message for errors
' IN   :
'------------------------------------------------------------------
Sub CheckCommand(errorMsg As String, consequence As Integer)
    If Len (errorMsg) > 0 Then
```

```
51          Select Case consequence
52          Case FATAL
53              RoseRTApp.WriteErrorLog "ERROR IN MATLAB : " & errorMsg
54              Beep
55              MsgBox "AN ERROR OCCURED :" & CRLF & errorMsg
56              Stop
57          Case Else
58              RoseRTApp.WriteErrorLog "MESSAGE FROM MATLAB : " & errorMsg
59          End Select
60      End If
61  End Sub
62
63
64  '-------------------------------------------------------------------
65  ' This function returns the port number associated with the string
66  ' passed in parameter. It uses the application data variable in Matlab.
67  ' IN    : direction As Integer, indicates if the port direction is IN or OUT
68  '         portName As string, the name of the port the program is looking for
69  ' OUT   : double, the port number in double format for compatibility with caller
70  ' Note : ad_uml must have been initialized in Matlab prior to calling this
71  '         function
72  '-------------------------------------------------------------------
73  Function getPortNumber(direction As Integer, portNameIn As String) As Double
74      Dim portHeight() As Double
75      Dim Mempty() As Double
76
77      If direction = PORT_IN Then
78          matlabServer.Execute "portHeightMat = ad_uml.fInputPortList.getData.
                getHeight;"
79          matlabServer.GetFullMatrix "portHeightMat", "base", portHeight, Mempty
80          For portID% = 1 To portHeight(0,0)
81              portName$ = matlabServer.Execute ("ad_uml.fInputPortList.
                    getCellData(" & (portID - 1) & ",0)")
82              If inStr(portName,portNameIn) Then
83                  Exit For
84              End If
85              'msgBox "Port Name for ID " & portID & " = " & portName
86          Next portID
87      Else
88          matlabServer.Execute "portHeightMat = ad_uml.fOutputPortList.getData.
                getHeight;"
89          matlabServer.GetFullMatrix "portHeightMat", "base", portHeight, Mempty
90          For portID% = 1 To portHeight(0,0)
91              portName$ = matlabServer.Execute ("ad_uml.fOutputPortList.
                    getCellData(" & (portID - 1) & ",0)")
92              If inStr(portName,portNameIn) Then
93                  Exit For
94              End If
95              'msgBox "Port Name for ID " & portID & " = " & portName
96          Next portID
97      End If
98
99      'The loop counter portID contains the corresponding value for the port
100     getPortNumber = portID
101
102 End Function
103
104
105 '-------------------------------------------------------------------
106 ' This function formats a string for M-File documentation.
107 ' IN    : String
108 '-------------------------------------------------------------------
109 Function RemoveNewline (inString As String) As String
110     Dim startPos As Integer
111     Dim crlfPos As Integer
112     Dim tempString As String
113
114     tempString = ""
```

```
115
116     crlfPos  = InStr(1, inString, CRLF, 1)
117     startPos = 1
118
119     While crlfPos <> 0
120         tempString = tempString & Mid$(inString, startPos, crlfPos - startPos)
                & " "
121         startPos = crlfPos + 2
122         crlfPos  = InStr(startPos, inString, CRLF, 1)
123     Wend
124     tempString = tempString & Mid$(inString, startPos)
125     RemoveNewline = tempString
126
127 End Function
128
129
130 '-------------------------------------------------------------------
131 ' This function formats a string for M-File documentation.
132 ' IN    : String
133 '-------------------------------------------------------------------
134 Function FormatDoc (inString As String) As String
135     Dim startPos As Integer
136     Dim crlfPos As Integer
137     Dim tempString As String
138
139     tempString = "% Documentation :" & CRLF
140
141     crlfPos  = InStr(1, inString, CRLF, 1)
142     startPos = 1
143
144     While crlfPos <> 0
145         tempString = tempString & "%    " & Mid$(inString, startPos, crlfPos -
              startPos) & CRLF
146         startPos = crlfPos + 2
147         crlfPos  = InStr(startPos, inString, CRLF, 1)
148     Wend
149     tempString = tempString & "%    " & Mid$(inString, startPos)
150     FormatDoc = tempString
151
152 End Function
153
154
155 '-------------------------------------------------------------------
156 ' This function formats a string for Matlab. New line and slash characters
157 ' are processed. The returned string is vector ready but does not include
158 ' the square brackets.
159 ' IN    : String
160 '-------------------------------------------------------------------
161 Function FormatString (inString As String) As String
162     Dim startPos As Integer
163     Dim crlfPos As Integer
164     Dim slashPos As Integer
165     Dim tempString As String
166     Dim tempString2 As String
167
168     tempString  = ""
169     tempString2 = ""
170
171     'First replace all CRLF by the appropriate command : sprint('\n')
172     crlfPos  = InStr(1, inString, CRLF, 1)
173     startPos = 1
174
175     While crlfPos <> 0
176         tempString = tempString & "'" & Mid$(inString, startPos, crlfPos -
              startPos) & "',sprintf('\n'),"
177         startPos = crlfPos + 2
178         crlfPos  = InStr(startPos, inString, CRLF, 1)
179     Wend
```

```
180     tempString = tempString & "'" & Mid$(inString, startPos) & "'"
181
182     'Then replace single slashes (/) by double slahes (//) as required by
            Matlab
183     slashPos  = InStr(1, tempString, "/", 1)
184     startPos = 1
185     While slashPos <> 0
186         tempString2 = tempString2 & Mid$(tempString, startPos, slashPos -
                startPos) & "//"
187         startPos = slashPos + 1
188         slashPos  = InStr(startPos, tempString, "/", 1)
189     Wend
190     tempString2 = tempString2 & Mid$(tempString, startPos)
191
192     FormatString = tempString2
193 End Function
194
195
196 '-------------------------------------------------------------------
197 ' Perform some extra configuration, e.g. initialize S-Function, subsystem
        attributes, etc.
198 ' IN    : ModelElement theCapsuleRole, the capsule role being analyzed
199 '         String subSystemName, the complete path name to the sub-system in the
        Simulink model
200 '-------------------------------------------------------------------
201 Sub ConfigureSFunctionName(theCapsuleRole As RoseRT.ModelElement, subSystemName
         As String)
202     Dim theCapsule As RoseRT.Capsule
203     Dim allOperations As RoseRT.OperationCollection
204     Dim theOperation As RoseRT.Operation
205     Dim matlabCommand As String
206
207     matlabCommand = "set_param('" & subSystemName & "','FunctionName','wrap_"
            & theCapsuleRole.Name & "')"
208     CheckCommand matlabServer.Execute(matlabCommand), FATAL
209
210     'Set theCapsule = theCapsuleRole.Capsule
211     'If theCapsule.Stereotype = "system" Then
212     '   Set allOperations = theCapsule.Operations
213     '   For opID% = 1 To allOperations.Count
214     '       Set theOperation = allOperations.getAt(opID%)
215             'Export the code to a standalone file that will be used by simulink
                    S-Function builder
216             'That standalone file should be specified by the File property of
                    the operation
217     '       matlabCommand = "set_param('" & subSystemName & "','FunctionName','
            wrap_" & theOperation.Name & "')"
218     '       RoseRTApp.WriteErrorLog "Configuring S-Function : " & matlabCommand
219     '       CheckCommand matlabServer.Execute(matlabCommand), FATAL
220             'Export the code
221             'Get X from theOperation.Files.getAt...
222             'Open X For Output As #1
223             'Print #1, theOperation.Code
224             'Close #1
225     '   Next opID
226     'End If
227 End Sub
228
229 '-------------------------------------------------------------------
230 ' Create an M-File for the given capsule. The file contains a all the
        attributes of the
231 ' capsule in the form attribute = value.
232 ' IN    : ModelElement theCapsuleRole, the capsule role being analyzed
233 '         String subSystemName, the complete path name to the sub-system in the
        Simulink model
234 '-------------------------------------------------------------------
235 Sub CreateMFile (theCapsuleRole As RoseRT.ModelElement, subSystemName As String
        )
```

```
236     Dim allAttributes As RoseRT.AttributeCollection
237     Dim anAttribute As RoseRT.Attribute
238     Dim subSystemMFileName As String
239
240     Set allAttributes = theCapsuleRole.Capsule.Attributes
241
242     subSystemMFileName = "C:\work\" & theCapsuleRole.Name & MFILE_EXT
243
244     Open subSystemMFileName For Output As #1
245     Print #1, "% Parameter file for sub-system " & theCapsuleRole.Name
246     If Len (theCapsuleRole.Documentation) > 0 Then
247         stringDoc$ = formatDoc(theCapsuleRole.Documentation)
248         Print #1, stringDoc
249     End If
250
251     Print #1, 'Print a blank line
252
253     For attID% = 1 To allAttributes.Count
254         Set anAttribute = allAttributes.GetAt(attID)
255         attributeDoc$ = anAttribute.Documentation
256         If Len(anAttribute.Documentation) > 0 Then
257             attributeDoc = "% " & RemoveNewline(attributeDoc)
258         End If
259         If (anAttribute.Type = "string") Or (anAttribute.Type = "char") Then
260             Print #1, anAttribute.Name & " = '" & anAttribute.InitValue & "';"
                    ,, attributeDoc
261         Else
262             Print #1, anAttribute.Name & " = " & anAttribute.InitValue & ";",,
                    attributeDoc
263         End If
264     Next attID
265
266     Close #1
267
268     'Add entry into top-level M-File
269     Open globalMFileName For Append As #1
270     Print #1, theCapsuleRole.Name & MFILE_EXT_1
271     Close #1
272 End Sub
273
274 '-------------------------------------------------------------------
275 ' Connect ports together
276 ' IN    : String systemName, system to which we are adding the ports
277 '         PortRole connectorEnd1, port role at the first end of the connector
278 '         PortRole connectorEnd2, port role at the second end of the connector
279 '-------------------------------------------------------------------
280 Sub ConnectPorts(systemName As String, connectorEnd1 As RoseRT.ModelElement,
    connectorEnd2 As RoseRT.ModelElement)
281     Dim capsuleRole1 As RoseRT.CapsuleRole
282     Dim capsuleRole2 As RoseRT.CapsuleRole
283     Dim allPorts    As RoseRT.PortCollection    'The port roles of the capsule
        role
284     Dim thePort     As RoseRT.Port
285     Dim theProtocol As RoseRT.Protocol          'Used to analyze in/out signals
286     Dim allInSignals  As RoseRT.SignalCollection    'In signals of a protocol
287     Dim allOutSignals  As RoseRT.SignalCollection   'Out signals of a protocol
288     Dim theSignal   As RoseRT.Signal
289     Dim theCapsule  As RoseRT.Capsule
290     Dim outSignalName As String 'Names of the ports used to create lines in
        Simulink
291     Dim inSignalName As String
292     Dim inPortNumber() As Double
293     Dim outPortNumber() As Double
294     Dim Mempty() As Double
295     Dim portHeightStr As String
296     Dim portHeight As Integer  'Height of the input/output port table of Matlab
         S-Function
297
```

```
298     'Find the capsule at the connector ends
299     If connectorEnd1.isClass("PortRole") Then
300         'End1 is a capsule role
301         Set capsuleRole1 = connectorEnd1.ParentCapsuleRole
302         'Retrieve the protocol signals
303         Set theCapsule = capsuleRole1.Capsule
304         Set allPorts = theCapsule.Structure.Ports
305
306         For portID% = 1 To allPorts.Count
307             Set thePort = allPorts.GetAt (portID)
308             If thePort.Name = connectorEnd1.Name Then
309                 Exit For
310             End If
311         Next portID
312     Else
313         'End1 is a port on the capsule's own boundary : a relay port
314         Set thePort = connectorEnd1
315     End If
316
317     'The reference protocol used here will be end1's
318     Set theProtocol = thePort.Protocol
319
320     If connectorEnd2.isClass("PortRole") Then
321         Set capsuleRole2 = connectorEnd2.ParentCapsuleRole
322     End If
323
324     'Route wires from connectorEnd1 to connectorEnd2, check if conjugated.
325     'Relay ports keep the original, non-conjugated signals
326     If connectorEnd1.isClass("PortRole") Then
327         If thePort.Conjugated Then
328             Set allOutSignals = theProtocol.InSignals
329             Set allInSignals  = theProtocol.OutSignals
330         Else
331             Set allOutSignals = theProtocol.OutSignals
332             Set allInSignals  = theProtocol.InSignals
333         End If
334     Else
335         If thePort.Conjugated Then
336             Set allOutSignals = theProtocol.OutSignals
337             Set allInSignals  = theProtocol.InSignals
338         Else
339             Set allOutSignals = theProtocol.InSignals
340             Set allInSignals  = theProtocol.OutSignals
341         End If
342     End If
343
344     For SignalID% = 1 To allOutSignals.Count
345         Set theSignal = allOutSignals.GetAt (SignalID)
346
347         If connectorEnd1.isClass("PortRole") Then
348             If capsuleRole1.Capsule.Stereotype = "system" Then
349                 'Connector End 1 is an S-Function
350                 ReDim outPortNumber(1,1)
351                 outPortNumber(0,0) = getPortNumber(PORT_OUT, connectorEnd1.Name
                        & "_" & theSignal.Name)
352             Else
353                 outSignalNameTmp$ = systemName & "/" & capsuleRole1.Name & "/"
                        & connectorEnd1.Name & "_" & theSignal.Name
354                 matlabCommand$ = "OutPortNumber = str2num(get_param('" &
                        outSignalNameTmp & "', 'Port'))"
355                 matlabServer.Execute matlabCommand
356                 matlabServer.GetFullMatrix "OutPortNumber","base",outPortNumber
                        , Mempty
357             End If
358             outSignalName =  capsuleRole1.Name & "/" & outPortNumber(0,0)
359         Else
360             outSignalName = connectorEnd1.Name & "_" & theSignal.Name & "/1"
361         End If
```

```
362
363         If connectorEnd2.isClass("PortRole") Then
364             If capsuleRole2.Capsule.Stereotype = "system" Then
365                 'Connector End 2 is an S-Function
366                 ReDim inPortNumber(1,1)
367                 inPortNumber(0,0) = getPortNumber(PORT_IN, connectorEnd2.Name
                        & "_" & theSignal.Name)
368             Else
369                 inSignalNameTmp$  = systemName & "/" & capsuleRole2.Name & "/"
                        & connectorEnd2.Name & "_" & theSignal.Name
370                 matlabCommand$ = "InPortNumber = str2num(get_param('" &
                        inSignalNameTmp & "', 'Port'))"
371                 matlabServer.Execute matlabCommand
372                 matlabServer.GetFullMatrix "InPortNumber","base",inPortNumber,
                        Mempty
373             End If
374             inSignalName  =  capsuleRole2.Name & "/" & inPortNumber(0,0)
375         Else
376             inSignalName  = connectorEnd2.Name & "_" & theSignal.Name & "/1"
377         End If
378
379         matlabCommand$ = "add_line('" & systemName & "','" & outSignalName & "
                ', '" & inSignalName & "','autorouting','on')"
380         'MsgBox matlabCommand
381         CheckCommand matlabServer.Execute(matlabCommand), FATAL
382     Next SignalID
383
384
385     'Then route wires from connectorEnd2 to connectorEnd1, check if conjugated
386     For SignalID% = 1 To allInSignals.Count
387         Set theSignal = allInSignals.GetAt (SignalID)
388
389         If connectorEnd2.isClass("PortRole") Then
390             If capsuleRole2.Capsule.Stereotype = "system" Then
391                 'Connector End 1 is an S-Function
392                 ReDim outPortNumber(1,1)
393                 outPortNumber(0,0) = getPortNumber(PORT_OUT, connectorEnd2.Name
                        & "_" & theSignal.Name)
394             Else
395                 outSignalNameTmp$ = systemName & "/" & capsuleRole2.Name & "/"
                        & connectorEnd2.Name & "_" & theSignal.Name
396                 matlabCommand$ = "OutPortNumber = str2num(get_param('" &
                        outSignalNameTmp & "', 'Port'))"
397                 matlabServer.Execute matlabCommand
398                 matlabServer.GetFullMatrix "OutPortNumber","base",outPortNumber
                        , Mempty
399             End If
400             outSignalName =  capsuleRole2.Name & "/" & outPortNumber(0,0)
401         Else
402             outSignalName = connectorEnd2.Name & "_" & theSignal.Name & "/1"
403         End If
404
405         If connectorEnd1.isClass("PortRole") Then
406             If capsuleRole1.Capsule.Stereotype = "system" Then
407                 'Connector End 1 is an S-Function
408                 ReDim inPortNumber(1,1)
409                 inPortNumber(0,0) = getPortNumber(PORT_IN, connectorEnd1.Name
                        & "_" & theSignal.Name)
410             Else
411                 inSignalNameTmp$  = systemName & "/" & capsuleRole1.Name & "/"
                        & connectorEnd1.Name & "_" & theSignal.Name
412                 matlabCommand$ = "InPortNumber = str2num(get_param('" &
                        inSignalNameTmp & "', 'Port'))"
413                 matlabServer.Execute matlabCommand
414                 matlabServer.GetFullMatrix "InPortNumber","base",inPortNumber,
                        Mempty
415             End If
416             inSignalName  =  capsuleRole1.Name & "/" & inPortNumber(0,0)
```

```
417         Else
418             inSignalName = connectorEnd1.Name & "_" & theSignal.Name & "/1"
419         End If
420
421         matlabCommand$ = "add_line('" & systemName & "','" & outSignalName & "
                ', '" & inSignalName & "','autorouting','on')"
422         'MsgBox  matlabCommand
423         CheckCommand matlabServer.Execute(matlabCommand), FATAL
424     Next SignalID
425 End Sub
426
427
428 '-----------------------------------------------------------------
429 ' Create input and output ports of a simulink capsule based on port roles
430 ' IN   : ModelElement theCapsuleRole, the reference model element that contains
        port roles
431 '         String subSystemName, the path to the subsystem
432 '         matlabServer (global), handle to the automation sever
433 '-----------------------------------------------------------------
434 Sub CreatePorts(theCapsuleRole As RoseRT.ModelElement, subSystemName As String)
435     Dim allPorts    As RoseRT.PortCollection    'The port roles of the capsule
            role
436     Dim thePort     As RoseRT.Port
437     Dim theProtocol As RoseRT.Protocol          'Used to analyze in/out signals
438     Dim allSignals  As RoseRT.SignalCollection  'In and Out signals of a
            protocol
439     Dim theSignal   As RoseRT.Signal
440     Dim theCapsule  As RoseRT.Capsule
441     Dim subSystemPosition As String
442     Dim inSignalCount As Integer                    'Store the signal number
            for positioning purpose
443     Dim outSignalCount As Integer                   'Store the signal number
            for positioning purpose
444     Dim plotCommand As String 'Used to create port labels for S-Functions
445     Dim resultMsg As String
446
447     Set theCapsule = theCapsuleRole.Capsule
448     Set allPorts = theCapsule.Structure.Ports
449
450     inSignalCount = 1
451     outSignalCount = 1
452
453     'LOG MESSAGE -----------------
454     RoseRTApp.WriteErrorLog "Creating ports of sub-system " & subSystemName
455
456     'Iterate through all the ports of the capsule
457     For PortID% = 1 To allPorts.Count
458
459         Set thePort     = allPorts.GetAt (PortID)
460         Set theProtocol = thePort.Protocol
461
462         'Analyze input signals, considering conjugated status
463         If thePort.Conjugated Then
464             Set allSignals = theProtocol.OutSignals
465         Else
466             Set allSignals = theProtocol.InSignals
467         End If
468
469         'Before creating signals for an S-Function, we need to set the total
                number of
470         'signals in the application data
471         If theCapsule.Stereotype = "system" Then
472             MatCmd$ = "ad_uml.fInputPortList.getData.setHeight(" & (allSignals.
                    Count + inSignalCount - 1) & ");"
473             CheckCommand matlabServer.Execute(matCmd), FATAL
474         End If
475
476         For SignalID% = 1 To allSignals.Count
```

```
477             Set theSignal = allSignals.GetAt (SignalID)
478             If theCapsule.Stereotype = "system" Then
479                 'Store the name of this input parameter in the application data
                        of the S-function.
480                 '"ad_uml" is initialized in caller function
481                 MatCmd$ = "ad_uml.fInputPortList.setCellData(" & (inSignalCount
                        - 1) &",0,'" & thePort.Name & "_" & theSignal.Name &"');"
482                 CheckCommand matlabServer.Execute(MatCmd), FATAL
483                 'Initialize remaining cells if above row 0 TO DO : Initialize
                        with proper type
484                 If (inSignalCount - 1) > 0 Then
485                     matlabServer.Execute "cell_data = ad_uml.fInputPortList.
                            getCellData(0,1)"
486                     matlabServer.Execute "ad_uml.fInputPortList.setCellData("
                            & (inSignalCount - 1) &",1,cell_data)"
487                     matlabServer.Execute "cell_data = ad_uml.fInputPortList.
                            getCellData(0,2)"
488                     matlabServer.Execute "ad_uml.fInputPortList.setCellData("
                            & (inSignalCount - 1) &",2,cell_data)"
489                     matlabServer.Execute "cell_data = ad_uml.fInputPortList.
                            getCellData(0,3)"
490                     matlabServer.Execute "ad_uml.fInputPortList.setCellData("
                            & (inSignalCount - 1) &",3,cell_data)"
491                     matlabServer.Execute "cell_data = ad_uml.fInputPortList.
                            getCellData(0,4)"
492                     matlabServer.Execute "ad_uml.fInputPortList.setCellData("
                            & (inSignalCount - 1) &",4,cell_data)"
493                     matlabServer.Execute "cell_data = ad_uml.fInputPortList.
                            getCellData(0,5)"
494                     matlabServer.Execute "ad_uml.fInputPortList.setCellData("
                            & (inSignalCount - 1) &",5,cell_data)"
495                     matlabServer.Execute "cell_data = ad_uml.fInputPortList.
                            getCellData(0,6)"
496                     matlabServer.Execute "ad_uml.fInputPortList.setCellData("
                            & (inSignalCount - 1) &",6,cell_data)"
497                 End If
498             Else
499                 MatCmd$ = "add_block('built-in/Inport', '" & subSystemName & "/
                        " & thePort.Name & "_" & theSignal.Name & "');"
500                 CheckCommand matlabServer.Execute(MatCmd), FATAL
501                 subSystemPosition = "[50 " & inSignalCount * 100 & " 90 " &
                        inSignalCount * 100 + 20 & "]"
502                 MatCmd$ = "set_param('" & subSystemName & "/" & thePort.Name &
                        "_" & theSignal.Name & "','Position'," & subSystemPosition
                         & ");"
503                 CheckCommand matlabServer.Execute(MatCmd), NORMAL
504             End If
505             inSignalCount = inSignalCount + 1 'Increment the global signal
                    count for positioning
506         Next SignalID
507
508         'Analyze output signals
509         If thePort.Conjugated Then
510             Set allSignals = theProtocol.InSignals
511         Else
512             Set allSignals = theProtocol.OutSignals
513         End If
514
515         'Before creating signals for an S-Function, we need to set the total
                number of
516         'signals in the application data
517         If theCapsule.Stereotype = "system" Then
518             MatCmd$ = "ad_uml.fOutputPortList.getData.setHeight(" & (allSignals
                    .Count + outSignalCount - 1) & ");"
519             CheckCommand matlabServer.Execute(MatCmd), FATAL
520         End If
521
522         For SignalID% = 1 To allSignals.Count
```

```
523             Set theSignal = allSignals.GetAt (SignalID)
524             If theCapsule.Stereotype = "system" Then
525                 'Store the name of this output parameter in the application
                        data of the S-function.
526                 '"ad_uml" is initialized in caller function
527                 MatCmd$ = "ad_uml.fOutputPortList.setCellData(" & (
                        outSignalCount - 1) &",0,'" & thePort.Name & "_" &
                        theSignal.Name &"');"
528                 CheckCommand matlabServer.Execute(MatCmd), FATAL
529
530                 'Initialize remaining cells if above row 0 TO DO : Initialize
                        with proper type
531                 If (outSignalCount - 1) > 0 Then
532                     matlabServer.Execute "cell_data = ad_uml.fOutputPortList.
                            getCellData(0,1)"
533                     matlabServer.Execute "ad_uml.fOutputPortList.setCellData("
                            & (outSignalCount - 1) &",1,cell_data)"
534                     matlabServer.Execute "cell_data = ad_uml.fOutputPortList.
                            getCellData(0,2)"
535                     matlabServer.Execute "ad_uml.fOutputPortList.setCellData("
                            & (outSignalCount - 1) &",2,cell_data)"
536                     matlabServer.Execute "cell_data = ad_uml.fOutputPortList.
                            getCellData(0,3)"
537                     matlabServer.Execute "ad_uml.fOutputPortList.setCellData("
                            & (outSignalCount - 1) &",3,cell_data)"
538                     matlabServer.Execute "cell_data = ad_uml.fOutputPortList.
                            getCellData(0,4)"
539                     matlabServer.Execute "ad_uml.fOutputPortList.setCellData("
                            & (outSignalCount - 1) &",4,cell_data)"
540                     matlabServer.Execute "cell_data = ad_uml.fOutputPortList.
                            getCellData(0,5)"
541                     matlabServer.Execute "ad_uml.fOutputPortList.setCellData("
                            & (outSignalCount - 1) &",5,cell_data)"
542                     matlabServer.Execute "cell_data = ad_uml.fOutputPortList.
                            getCellData(0,6)"
543                     matlabServer.Execute "ad_uml.fOutputPortList.setCellData("
                            & (outSignalCount - 1) &",6,cell_data)"
544                 End If
545
546             Else
547                 MatCmd$ = "add_block('built-in/Outport', '" & subSystemName & "
                        /" & thePort.Name & "_" & theSignal.Name & "');"
548                 CheckCommand matlabServer.Execute(MatCmd), FATAL
549                 subSystemPosition = "[500 " & outSignalCount * 100 & " 540 " &
                        outSignalCount * 100 + 20 & "]"
550                 MatCmd$ = "set_param('" & subSystemName & "/" & thePort.Name &
                        "_" & theSignal.Name & "','Position'," & subSystemPosition
                         & ");"
551                 CheckCommand matlabServer.Execute(MatCmd), NORMAL
552             End If
553             outSignalCount = outSignalCount + 1 'Increment the global signal
                    count for positioning
554         Next SignalID
555     Next PortID
556
557     'Check if there is at least 1 input and 1 output signal defined for the S-
            function
558     If (inSignalCount = 1) Or (outSignalCount = 1) Then
559         msgBox "ERROR!" & CRLF & "The S-Function needs at least one input
                signal and" & CRLF _
560                     & "one output signal. Check protocol definition of all
                            ports of the" & CRLF _
561                     & "S-Function in UML! Generation will stop here."
562         Stop
563     End If
564 End Sub
565
566
```

```
567  '-------------------------------------------------------------------
568  ' Compute the position of an element in the Simulink model window
569  ' IN   : theElement, the diagram element we want to position
570  ' OUT  : string, the array required by the position property
571  ' NOTE : sring Is of the form "[left top right bottom]"
572  '-------------------------------------------------------------------
573  Function ComputeElementPosition(theElement As RoseRT.ViewElement) As String
574      leftPosition%    = (theElement.XPosition - (theElement.Width  / 2)) \ SCALE
             ' \ = Integer division
575      topPosition%     = (theElement.YPosition - (theElement.Height / 2)) \ SCALE
576      rightPosition%   = (theElement.XPosition + (theElement.Width  / 2)) \ SCALE
577      bottomPosition%  = (theElement.YPosition + (theElement.Height / 2)) \ SCALE
578      'Assign result
579      ComputeElementPosition = "[" & leftPosition & " " & topPosition & " " &
             rightPosition & " " & bottomPosition & "]"
580  End Function
581
582  '-------------------------------------------------------------------
583  ' Anaalyze the structure of a capsule and extract capsule roles
584  '-------------------------------------------------------------------
585  Sub ParseStructure(theCapsuleStructureDiagram As RoseRT.Diagram, systemName As
         String)
586      'Model elements
587      Dim theCapsuleStructure As RoseRT.CapsuleStructure
588      Dim theCapsuleRole As RoseRT.CapsuleRole
589
590      Dim allViewElements As RoseRT.ViewElementCollection
591      Dim theViewElement As RoseRT.ViewElement
592
593      Dim subSystemClassifierRoles As RoseRT.ClassifierRoleCollection
594      Dim theClassifierRole As RoseRT.ClassifierRole
595
596      Dim allConnectors As RoseRT.ConnectorCollection
597      Dim theConnector As RoseRT.Connector
598      Dim connectorEnd1 As RoseRT.ModelElement 'Used to retrieve port or port
             roles
599      Dim connectorEnd2 As RoseRT.ModelElement
600
601      Dim theModelElement As RoseRT.ModelElement
602      Dim tempModelElement As RoseRT.ModelElement 'Used to analyze ports and port
              roles
603
604      Dim isSFunction As Boolean ' TRUE if the capsule being analyzed is of the s
             -function stereotype
605
606      'Utility variables
607  '   Dim systemName As String
608      Dim subSystemName As String
609      Dim subSystemPosition As String
610      Dim resultMsg As String
611
612      Set allViewElements = theCapsuleStructureDiagram.ViewElements
613
614      'Needed to retrieve connectors: retrieve the top capsule structure
615      Set theCapsuleStructure = theCapsuleStructureDiagram.ParentModelElement.
             Structure
616      Set allConnectors = theCapsuleStructure.Connectors
617
618      'Retrieve the current system name
619   '  systemName = theCapsuleStructureDiagram.ParentModelElement.Name
620
621      'LOG MESSAGE ------------
622      RoseRTApp.WriteErrorLog "Parsing system " & systemName
623
624      'Iterate through all the view elements of the diagram
625      For ClsID% = 1 To allViewElements.Count
626
627          Set theViewElement = allViewElements.GetAt (ClsID)
```

```
628         If theViewElement.HasModelElement Then
629
630             If theViewElement.IsClass ("CapsuleRoleView") Then
631
632                 'Retrieve the capsule role associated with this CapsuleRoleView
633                 Set theModelElement = theViewElement.ModelElement
634
635                 'Test to see if capsule is of stereotype "system"
636                 If theModelElement.Capsule.Stereotype = "system" Then
637                     isSFunction = TRUE
638                 Else
639                     isSFunction = FALSE
640                 End If
641
642                 subSystemName = systemName & "/" & theViewElement.name
643
644                 'LOG MESSAGE -------------
645                 RoseRTApp.WriteErrorLog "Adding sub-system " & subSystemName
646
647                 'Retrieve block position and size : [left top right bottom
                        ] !!!!!!
648                 subSystemPosition = computeElementPosition(theViewElement)
649                 If isSFunction Then
650                     resultMsg = matlabServer.Execute("add_block('simulink/User-
                            Defined Functions/S-Function Builder','" &
                            subSystemName & "')")
651                     ConfigureSFunctionName theModelElement, subSystemName
652                     'For S-Function blocks, retrieve the application data for
                            further configuration
653                     'Application data is stored locally in Matlab
654                     matlabServer.Execute "sfunc_handle = get_param('" &
                            subSystemName & "', 'Handle');"
655                     matlabServer.Execute "ad_uml = sfunctionwizard(sfunc_handle
                            , 'GetApplicationData');"
656                 Else
657                     resultMsg = matlabServer.Execute("add_block('built-in/
                            SubSystem', '" & subSystemName & "')")
658                 End If
659
660                 'ERROR MESSAGE -----------
661                 If Len(resultMsg) > 0 Then
662                     MsgBox "An error occured during the creation of a block in
                            Matlab : " & CRLF & resultMsg
663                     Stop
664                 End If
665
666                 matlabServer.Execute "set_param('" & subSystemName & "','
                        Position'," & subSystemPosition & ")"
667
668                 elementDoc$ = FormatString(theModelElement.Documentation)
669
670                 matlabServer.Execute "set_param('" & subSystemName & "','
                        Description',[" & elementDoc & "])"
671
672                 'Create ports of a given subsystem
673                 CreatePorts theModelElement,subSystemName
674
675                 'Create subsystem M-File
676                 CreateMFile theModelElement,subSystemName
677
678                 'Configure extras of the subsystem
679                 'OBSOLETE ConfigureExtras theModelElement, subSystemName
680
681                 'Done with structure and ports of this capsule. Now
682                 'check for entry into recursion if not an S-Function or
                        complete
683                 'the S-Function by building it
684                 If isSFunction Then
```

```
685                     'Build the S-Function block to update port and code info
686                     buildMsg$ = matlabServer.Execute ("sfunctionwizard(
                            sfunc_handle, 'Build', ad_uml);")
687                     RoseRTApp.WriteErrorLog "Build result : " & buildMsg
688                 Else
689                     Set subSystemClassifierRoles = theModelElement.Capsule.
                            Structure.ClassifierRoles
690                     If subSystemClassifierRoles.Count > 0 Then
691                         ParseStructure theModelElement.Capsule.Structure.
                                Diagram, subSystemName
692                     End If
693                 End If 'If Not isSFunction Then
694             End If 'If theViewElement.IsClass ("CapsuleRoleView") Then
695         End If 'If theViewElement.HasModelElement Then
696
697         If theViewElement.IsClass("NoteView") Then
698             If theViewElement.GetNoteViewType() = 2 Then
699                 theViewElementName$ = formatString(theViewElement.Text)
700                 subSystemName = "'" & systemName & "/'," & theViewElementName
701                 'Note : strangely enough, the x-axis of notes has twice the
                        density than that for standard blocks.
702                 'See example at http://www.mathworks.com/access/helpdesk/help/
                        toolbox/simulink/ug/creating_model14.shtml
703                 annotationPosition$ = "[" & ((theViewElement.XPosition \ SCALE)
                        * 2) _
704                                         & ",0,0," _
705                                         & (theViewElement.YPosition \ SCALE) _
706                                         & "]"
707                 matlabCommand$ = "add_block('built-in/Note',[" & subSystemName
                        & "],'Position',"& _
708                                 annotationPosition & ",'DropShadow','on')"
709
710                 CheckCommand matlabServer.Execute(matlabCommand), NORMAL
711             End If
712         End If
713     Next ClsID
714
715     'Once all subsystems are created, iterate through connectors and wire the
            ports together
716     For ConnID% = 1 To allConnectors.Count
717         Set theConnector = allConnectors.GetAt (ConnID)
718         'Check if connector ends are ports or port roles
719         Set connectorEnd1 = theConnector.PortRole1
720         If connectorEnd1 Is Nothing Then
721             Set connectorEnd1 = theConnector.Port1
722 '           msgBox "Detected connector end 1 is port"
723         End If
724
725         Set connectorEnd2 = theConnector.PortRole2
726         If connectorEnd2 Is Nothing Then
727             Set connectorEnd2 = theConnector.Port2
728 '           msgBox "Detected connector end 2 is port"
729         End If
730
731         'Connect ports
732         'LOG MESSAGE ---------------
733         RoseRTApp.WriteErrorLog "Adding connectors in " & systemName
734         ConnectPorts systemName, connectorEnd1, connectorEnd2
735     Next ConnID
736
737 End Sub
738
739 '-------------------------------------------------------------------
740 ' Generates a Simulink model file
741 '-------------------------------------------------------------------
742 Sub GenerateSimulinkModel(theModel As RoseRT.Model)
743     Dim theDiagram As RoseRT.Diagram
744     Dim topCapsule As RoseRT.ModelElement
```

```
745     Dim modelName As String
746     Dim simulinkFileName As String
747
748     Set theDiagram = theModel.GetActiveDiagram ()
749     If theDiagram Is Nothing Then
750         MsgBox "No active structure diagram. Please open a structure diagram in
                the toolset."
751         Exit Sub
752     End If
753
754     'Get the name of the capsule the diagram belongs to
755     Set topCapsule = theDiagram.ParentModelElement
756     modelName = topCapsule.Name
757
758     Begin Dialog UserDialog ,,276,104,"Simulink Generator"
759         PushButton 164,72,40,14,"Generate",.Generate
760         CancelButton 208,72,40,14
761         Text 12,6,204,18,"This tool will generate a Simulink model from the
                active structure diagram.",.Text1,,,ebBold
762         TextBox 84,42,176,12,.TextBoxFileName
763         Text 12,44,44,8,"Output File :",.Text3
764         Text 12,28,68,8,"Reference Capsule :",.Text4
765         Text 84,28,148,8,modelName,.TextSystemName
766     End Dialog
767
768     Dim StartGenerationDlg As UserDialog
769
770     'Initialize default file name
771     StartGenerationDlg.TextBoxFileName = "C:\work\" & modelName & ".mdl"
772     r% = Dialog (StartGenerationDlg,1,0)
773
774     If r% = 1 Then
775         'LOG MESSAGE ------------
776         RoseRTApp.WriteErrorLog "Initializing Matlab handle..."
777
778         'Initialize the OLE object. Handle is declared global at the top of
                this script
779         Set matlabServer = CreateObject("matlab.application")
780         matlabServer.MaximizeCommandWindow
781
782         'Open Simulink library to add simulink blocks (specified with add_block
                (simulink/...))
783         matlabServer.Execute "simulink"
784
785         'LOG MESSAGE -----------
786         RoseRTApp.WriteErrorLog "Starting Simulink model generation for top
                system " & modelName
787
788         simulinkFileName = StartGenerationDlg.TextBoxFileName
789
790         'Create top-level Simulink system
791         matlabServer.Execute "new_system('" & modelName & "')"
792
793         'Create top-level M-File
794         globalMFileName = "C:\work\" & modelName & MFILE_EXT
795         Open globalMFileName For Output As #1
796         Print #1, "% Main parameter file for model " & modelName
797         Print #1, "% Parameters are generated from capsule attributes."
798         Print #1, "% Each capsule / subsystem has its own M-File."
799         Print #1, 'Empty line
800         Close #1
801
802         'To do : replace file name by working directory
803         ParseStructure theDiagram, modelName
804
805         'Once the system is created, open it for user validation
806         matlabServer.Execute "open_system('" & modelName & "')"
807
```

```
808             'Save the opened system in the working directory
809             matlabServer.Execute "save_system"
810             MsgBox "System saved, click OK to close Matlab"
811             matlabServer.Execute "close_system"
812             matlabServer.Quit
813             'LOG MESSAGE ----------------
814             RoseRTApp.WriteErrorLog "Generation completed"
815         End If
816     End Sub
817
818     '---------------------------------------------------------------------
819     'Main
820     '---------------------------------------------------------------------
821     Sub Main
822         GenerateSimulinkModel RoseRTApp.CurrentModel
823     End Sub
```

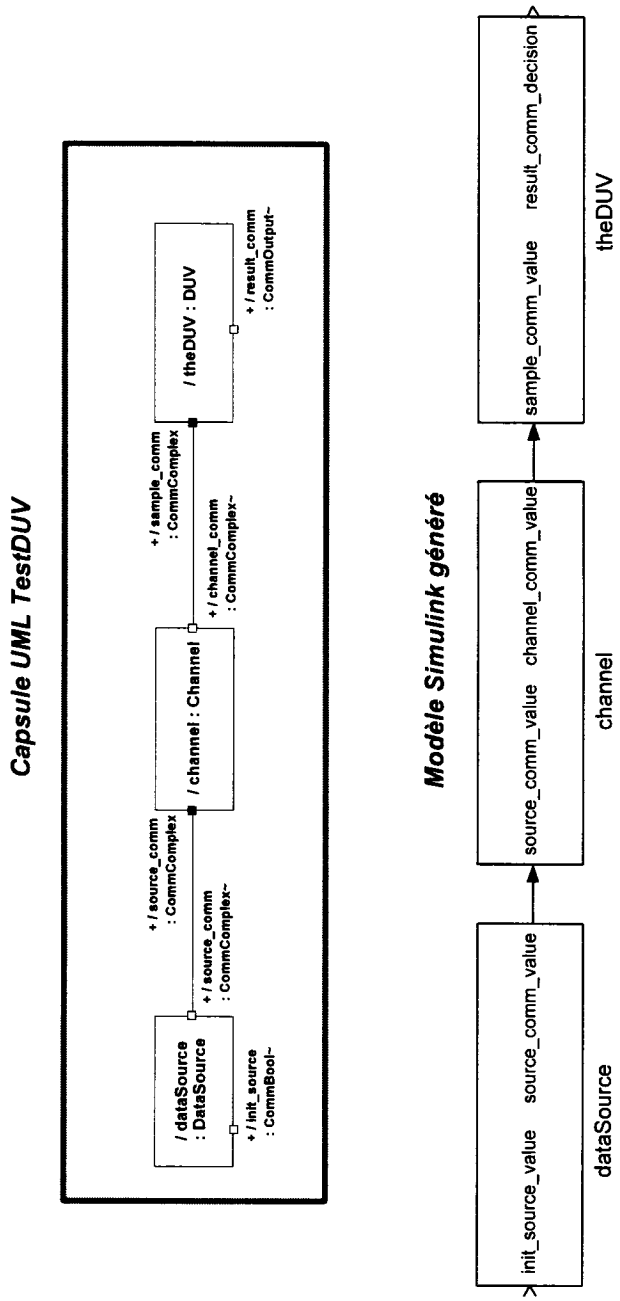

FIG. III.1: Exemple de transformation automatisée d'un modèle UML en un modèle Simulink